HyperSpeed XL/XE V2.1

Vorläufige Dokumentation

(Änderungen vorbehalten)

by Digital Force

2.5.1996

Vorwort

Wozu HyperSpeed XL/XE?

Mit HyperSpeed XL/XE (im folgenden nur noch HyperSpeed genannt) habe ich versucht, eine Brücke vom damaligen technischen Stand (als die 8Bit ATARIs entwickelt wurden) zur Gegenwart zu schlagen — wenigstens Ansatzweise. Nur 'Ansatzweise', nicht weil etwa nicht mehr machbar ist, sondern weil HyperSpeed wenigstens noch halbwegs bezahlbar sein soll. Die Entwicklung von HyperSpeed ergab sich nähmlich aus der Tatsache heraus, daß ich mit meinen Projekten, die ich mit Hilfe des alten XL/XEs realisieren wollte relativ schnell an die Grenze des machbaren gelangt bin. Also mußte sich irgendetwas ändern. Es gibt zwar schon diverse CPU-Erweiterungen, aber diese sind meines Wissens kaum zu einer brauchbaren Geschwindigkeitssteigerung zu gebrauchen. Bei HyperSpeed handelt es sich auch weniger um eine Erweiterung, sondern eher um einen neuen Rechner, der den alten XL/XE sozusagen 'on the fly' bedienen kann. Über Sinn oder Unsinn von HyperSpeed läßt sich streiten...
HyperSpeed ist als solches noch kein fertiges Produkt, sondern bietet derzeit eine mehr oder weniger solide Basis für Weiterentwicklungen. Wird in dieser Dokumentation von einem Betriebssystem gesprochen, so existiert dieses momentan praktisch nicht oder nur teilweise.

Zu dieser Dokumentation

Ich habe versucht die Dokumentation möglichst verständlich zu gestalten. Allerdings gehe ich von einem gewissen Grundverständnis und ein wenig Hardwarekenntnis aus. Ich empfehle aber auch dem Laien alles einmal durchzulesen, weil dadurch einige Sachen sicherlich besser verstanden werden. Trotzdem wird es manchmal auch Passagen geben, die auch von Hardware-Experten schwierig nachzuvollziehen sein werden. Das liegt daran, daß teilweise aus Performancegründen eine unübliche aber dennoch legale Richtung eingeschlagen wurde. Denn Modelle sind zwar schön und gut und meist einfach zu handhaben, aber bei der damit verbundenen Abstraktion geht leider auch der direkte Bezug zum niedrigsten Level — der Realität — verloren. Dies hat (in vielen Fällen) zur Folge, daß nicht mehr direkt am Problem gearbeitet werden kann woraus dann wieder Leistungsverluste in Kauf zu nehmen sind. Der Entwicklungs(zeit)aufwand steigt dadurch aufgrund der Komplexität überproportional an — obwohl das Ergebnis hinterher meist kompakter und einfacher erscheint. Das ist in etwa vergleichbar mit der Programmierung in einer Hochsprache oder Assembler.

Inhaltsverzeichnis

Kapitel 1

Ein grober Überblick

Abbildung 1.1 zeigt die allgemeine Architektur von HyperSpeed mit entsprechenden Beziehungen zwischen den einzelnen Komponenten.

1.1 CPU

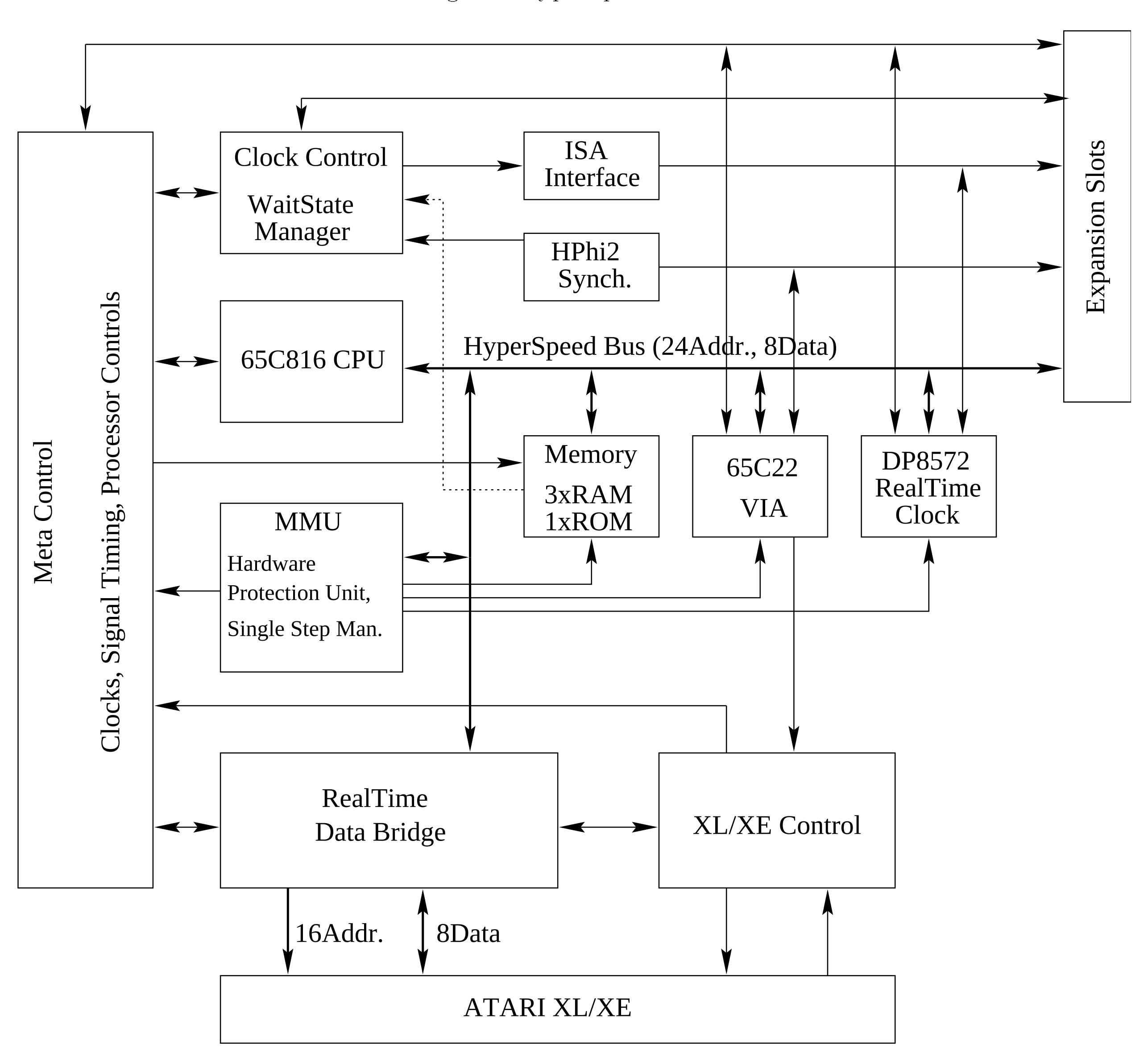
HyperSpeed besitzt wie gesagt eine 16Bit 65C816 CPU von WDC welche laut Datenblatt mit maximal 14MHz betrieben werden kann. Sie macht zwar auch ein bischen mehr mit, aber dann kann man nicht mehr 100%ige Funktionalität garantieren. Im Gegensatz zur 6502 oder 65C02 kann die 65C816 16MB Speicher linear verwalten. Die CPU an sich ist gegenüber der alten 6502 recht leistungsfähig, kann sich aber bei weitem noch nicht mit einem Pentium oder ähnlichen Boliden messen. Der größte Schwachpunkt dürfte wohl hier am nur 8Bit breiten Datenbus liegen. Das reißt die Performance doch ganz schön herunter. Moderne Architekturkonzepte wie Prefetching, Befehlspipelining, interner Cache usw. sucht man hier leider auch vergeblich. Weitere Daten über die CPU sind im Anhang beschrieben.

1.2 Speicher

1.2.1 RAM

HyperSpeed wird on Board ausschließlich mit statischen RAMs betrieben. Dafür gibt es drei Bänke bzw. Sockel. Zwei davon können jeweils 128kx8 im 32pol. SOJ-Gehäuse (Center-Power Ausführung) aufnehmen und ein Sockel kann RAMs von 64kx8 bis 512kx8 im 32pol. DIP Gehäuse (herkömmliche JEDEC Ausfürung) aufnehmen. Insgesamt sind also maximal 768kB RAM on Board möglich. Die Minimalkonfiguration sind 128kB, ansonsten macht das ganze wenig Sinn. Für jeden RAM-Chip lassen sich jeweils ein halber oder ein ganzer Warteschritt einstellen (derzeit noch über Jumper). Wenn man einen RAM ohne Warteschritte betreiben will, muß dieser theoretisch eine maximale Zugriffszeit von 15ns haben (bei 14MHz). Bei meinen Tests lief aber auch ein 20ns RAM problemlos ohne Warteschritte. Aber eine Garantie kann man wie gesagt keine geben. Pro eingestelltem halben Warteschritt erhöht sich die maximale Zugriffszeit um ca.35ns (bei 14MHz), so daß auch RAMs bis hoch zu 85ns Zugriffszeit sicher on Board verwendet werden können.
Wird mehr Speicher benötigt als die 768kB, dann läßt sich mehr oder weniger einfach zumindest aus statischen RAMs eine Speichererweiterung herstellen. Diese kann man dann entweder 'Huckepack' auf

Abbildung 1.1: HyperSpeed Architektur



HyperSpeed oder als Einsteckkarte konzipieren. Etwas aufwendiger wird das Einbinden von dynamischen RAMs (etwa SIMMs).

1.2.2 ROM

Nach dem Einschalten oder nach einem Reset bootet HyperSpeed nicht das XL/XE OS, sondern das HyperSpeed BIOS. Das ist notwendig, um HyperSpeed nach einem Reset die volle Kontrolle über den XL/XE zu geben. Das BIOS befindet sich auf einem on Board EPROM oder Flash. Darin sind wesentliche Funktionen zur Steuerung von HyperSpeed verankert — mehr aber momentan noch nicht. Es ist auch zu raten, nur diese Funktionen zu verwenden (um eine maximale Kompatibilität zu eventuellen weiterentwicklungen von HyperSpeed zu wahren). Die direkte Manipulation von HyperSpeed-spezifischen Hardwareregistern entfällt also. Das kostet zwar etwas Zeit, aber die sollte man sich doch nehmen, auch dann, wenn nur ein Bit geändert werden muß.
Es wurde bereits erwähnt, daß das ROM in Form eines EPROMs oder Flashs in Erscheinung tritt. Auf dem Board befindet sich ein 32poliger Sockel, der wahlweise einen 28poligen oder 32poligen ROM im DIP-Gehäuse nach JEDEC aufnehmen kann. Bei der EPROM-Variante sind Kapazitäten von 16kB bis hin zu 1MB steckbar (alles in nur einem Chip). Eine etwas teurere, aber ungleich flexiblere Alternative zu einem EPROM ist der Einsatz eines Flash-ROMs. Hier kommen vorzugsweise Flashs der Serie 29F... von AMD zum Einsatz. Diese sind in Sektoren fester oder unterschiedlicher Länge unterteilt, welche einzeln (Sektorweise) gelöscht und (fast) wie ein EEPROM Byteweise beschrieben werden können. Zum Löschen sowie zum Beschreiben werden allerdings spezielle Algorithmen benötigt, die aber vom BIOS mit zur Verfügung gestellt werden. Damit bietet sich die Möglichkeit, eigene Routinen direkt im ROM abzulegen. Die ROM Kapazität beim Einsatz eines Flash-ROMs schränkt sich allerdings auf einen Bereich von 128kB bis 512kB ein (zumindest in der Normalausführung).
Für den ROM-Chip sind übrigens 0.5, 1.0, 1.5 und 2.0 Warteschritte einstellbar. Bei 14MHz CPU-Takt und 2.0 Warteschritten ergibt sich eine Zugriffszeit von etwa 155ns. Das dürfte eigentlich ausreichen...

1.3 WaitState Manager

Der WaitState Manager spielt eine zentrale Rolle im Taktmanagement. Dieser manipuliert direkt den CPU Takt und wird außer zum 'Taktziehen' für die Speicherzugriffe noch von anderen HyperSpeed-internen Einheiten benutzt. Dem Nutzer stehen dabei 4 Leitungen zur Verfügung, die sich normalerweise auf 0.5, 1.0, 1.5 und 2.0 WaitStates beziehen. Es ist sicher ungewöhnlich, und aus der PC-Welt gänzlich unbekannt, daß die Warteschritte in halben Schritten, und nicht in ganzen zugeteilt werden. Diese Tatsache ergab sich aus folgender Überlegung:
Ein 20ns RAM ohne Warteschritte würde bei 14MHz theoretisch schon zu langsam sein (15ns notwendig). Es müsste also ein Warteschritt eingelegt werden. Das Resultat wäre eine effektive Zugriffszeit von ca. 15ns+70ns=85ns. Das ist natürlich paradox. Denn ein 20ns RAM kostet in etwa das doppelte eines 70ns RAMs. In der Praxis würde es aber in dem Beispiel keinen Unterschied machen, ob ein 20ns oder 70ns RAM verwendet wird. Es bedarf sicher keiner weiterer Erklärung, was da die Einführung von halben Warteschritten für Vorteile bietet. Noch besser wäre die Einführung von viertel oder achtel Warteschritten, denn bei dem Beispiel muß ja bloß wegen 5ns ein halber Warteschritt eingelegt werden. Aber dazu müßte ja der Systemtakt entsprechend erhöht werden, was wieder andere Probleme nach sich zieht. Außerdem ist 100Hier noch eine kurze Erklärung des Prinzips des WaitState Managers. Wird bei einem Speicherzugriff ein Warteschritt ausgelöst, dann wird der CPU-Takt entsprechend gestreckt. So ist zum Beispiel ein Zyklus mit einem halben Warteschritt 1.5 mal so lang wie ein normaler.

1.4 HPhi2 Synchronizer

HPhi2 steht hier für 'Half Phi2', also für den halben CPU-Takt. Genaugenommen handelt es sich eigentlich nicht um den halben CPU-Takt, sondern um den viertel Systemtakt — der seinerseits beim doppelten CPU-Takt liegt. Das klingt alles etwas verwirrend und ist für den Programmierer ohnehin ohne Bedeutung. Eine detailliertere Beschreibung befindet sich in der Technischen Dokumentation. Es soll nur kurz das Prinzip erläutert werden.
Das Bustiming Taktsynchroner Peripheriebausteine der 6500er Serie (wie sie sich sämtlich auch im XL/XE befinden) wird normalerweise direkt über den Takt der CPU abgewickelt. Auf HyperSpeed wird auch so ein Chip verwendet, und zwar der 65C22 oder auch VIA (Versatile Interface Adapter). Dieser Chip ist sozusagen eine Luxusausführung des PIAs im XL/XE. Er bietet außer den zwei 8Bit Ports noch ein paar andere Sachen, wie z.B. zwei 16Bit Timer. Würde nun dieser VIA direkt über den CPU-Takt versorgt, dann gibt es zwei Probleme:

- Der VIA ist momentan nur für einen Takt von 10MHz erhältlich.
- Da der CPU-Takt durch den WaitState Manager direkt manipuliert wird, würden die Timer des VIAs praktisch falsch gehen.

Aus dieser Tatsache ergab sich die Idee mit dem halben CPU-Takt, der aber nicht aus dem CPU-Takt, sondern aus dem Systemtakt — praktisch noch vor dem WaitState Manager — abgeleitet wird. Nun passen aber die beiden Takte (CPU und VIA) nicht so einfach zusammen. Aus diesem Grund ist der HPhi2 Synchronizer notwendig. Bei einem Zugriff auf einen Chip, der über HPhi2 betrieben wird, wird nun der CPU-Takt entsprechend zurechtgebogen bzw. gezogen. In der Realität ist diese Einheit direkt mit im WaitState Manager integriert (siehe Technische Dokumentation).

1.5 ATARI RealTime DataBridge

Hierbei handelt es sich um die eigentliche Innovation, sozusagen dem Herzstück von HyperSpeed. Diese sog. asynchronous Data Bridge bildet eine mehr oder weniger intelligente Schnittstelle zum XL/XE. Die DataBridge leitet ATARI-Zugriffe der HyperSpeed CPU (bzw. eines aktiven HyperSpeed Bus-Devices) über einen Zwischenbuffer an den XL/XE weiter. Dementsprechend wird in Read- und Writezugriffe unterschieden.
Sollte parallel zu HyperSpeed noch die alte ATARI-CPU laufen, so wird diese automatisch gestoppt.

1.5.1 Read from ATARI

Die 16Bit ATARI-Adresse, von der gelesen werden soll, wird an die DataBridge übergeben und die 65C816 wird erst einmal angehehalten. Die DataBridge versucht nun den nächstmöglichen freien ATARI-Zyklus auszunutzen, um das gewünschte Byte zu lesen. Ist dies geschehen dann wird die 65C816 wieder losgelassen und das gelesene Byte wird an die CPU übergeben.

1.5.2 Write to ATARI

Zuerst wird überprüft, ob der Zwischenbuffer voll ist. Ist der Buffer voll (also läuft noch ein älterer Schreibzugriff), dann wird die 65C816 vorerst gestoppt. Ist der Buffer leer oder wird gerade leer,

dann übernimmt die DataBridge die ATARI-Adresse und das zu schreibende Byte. Damit hat sich für die 65C816 ein Schreibzugriff auf den ATARI erledigt. Die DataBridge versucht nun auch das Byte schnellstmöglich an den XL/XE loszuwerden und den Buffer wieder auf 'leer' zu setzen. Schreibzugriffe laufen also nahezu ungebremst ab. Um einen maximalen Datendurchsatz zu erhalten, sollten die Schreibzugriffe aber nicht unmittelbar aufeinander folgen, weil ja der Buffer nicht so schnell geleert werden kann. Es fällt vielleicht auf, daß es sich hierbei um die triviale Form einer Write-Pipeline handelt - und zwar um einen FIFO (First In First Out) der Tiefe 1. Tatsächlich könnte man die Performance der DataBridge noch um einiges steigern, indem man die Tiefe des Buffers erhöht. Das habe ich allerdings aus Kostenngründen vorerst unterlassen...

1.6 ATARI Hardware Protection Unit

Mit dieser Komponente ist es möglich Zugriffe auf die ATARI Hardwareregister ($00D000 – $00D7FF) abzufangen. In der Praxis sieht das dann so aus, daß der Maschinenbefehl, der auf ein ATARI Hardwareregister zugreift regelrecht abgebrochen wird, noch bevor das Byte geschrieben bzw. gelesen wird. Unmittelbar darauf wird vom OS entschieden, ob der Zugriff gestattet wird, oder ob er nur teilweise 'durchgelassen' wird usw. Die Möglichkeiten sind dabei unbegrenzt. Von einer hundertprozentigen RAM-Disk Emulation (egal nach welchem Standard) bis evtl. hin zu mehreren virtuellen ATARIs, die parallel laufen. Das Problem ist nur die Software-Steuerung.
Noch besser wäre es, wenn auch Zugriffe auf den VIDEO RAM abgefangen werden können. Aber um die Kosten nicht noch mehr explodieren zu lassen ist dies momentan noch nicht möglich. Vom Prinzip her ist es aber kein Problem, dieses in Form einer Einsteckkarte nachzurüsten.

1.7 Speicheraufteilung

Hier waren auch einige Hürden zu meistern. Nicht in Bezug auf HyperSpeed direkt, sondern in Bezug auf das ATARI-Mapping. Also wie wird der ATARI-Speicher in den Adressraum der 65C816 eingeblendet? Diese Betrachtungen sind besoders in der Hinsicht notwendig, wie alte Programme für den XL/XE beschleunigt werden können. Aus diesem Grund ist der logische ATARI-Speicherbereich (die ersten 64kB im Gesamtadressraum von 16MB) in mehrere Segmente aufgeteilt worden. Für jedes einzelne Segment kann nun ausgewählt werden, ob sich an dieser Stelle Speicher aus dem XL/XE oder neuer, schneller Speicher befindet. Weitere Einzelheiten befinden sich in der Technischen Dokumentation.

1.8 RealTime Clock

Dies ist ein Uhrenbaustein ähnlich dem, der sich auch in neueren Rechnern befindet. Neben der Uhren- und Datumsfunktion (Jahr, Monat, Tag, Wochentag, Stunde, Minute, Sekunde, 1/100Sekunde, DatumsvergleichsRAM) enthält das Teil auch noch 31 Bytes und ein paar Bits RAM. Da dieser Baustein batteriegepuffert ist, behält der RAM die gespeicherten Daten wenn HyperSpeed ausgeschaltet wird. In diesem Fall läuft natürlich auch die Uhr weiter.
Da der RealTime Clock Chip relativ teuer ist, ist er optional bestückbar.

1.9 ISA Interface

Hierbei handelt es sich um kein vollständiges ISA-Interface, sondern nur um einen kleinen Subset — und zwar nur um die beiden Leitungen, die das Lesen und Schreiben steuern (sozusagen das Nötigste). Diese Leitungen müssen ein ganz bestimmtes Timing aufweisen. Über das ISA Interface können asynchrone Bausteine relativ einfach und Timing-unkritisch eingebunden werden. Dieses Interface wird zum Beispiel zur Ansteuerung des Uhrenbausteins verwendet.

1.10 Single Step Manager

Diese Einheit kann in Zusammenhang mit einem (noch zu entwickelnden...) Debugger verwendet werden, um ein Programm schrittweise von der 65C816 abarbeiten zu lassen. Das läuft sinngemäß so ab, daß nach der Auslösung einer Sequenz nach einer festgelegten Anzahl von OpCode Fetchs der Programmablauf unterbrochen wird. Eine bessere Lösung wäre zwar der Einsatz von Break Points, aber das hätte die Kosten von HyperSpeed wieder um einiges angehoben...

Kapitel 2

Technische Dokumentation

Im folgendem wird es um die technischen Einzelheiten der verschiedenen Komponenten gehen. Dabei geht es nicht nur darum, wie diese Komponenten genutzt werden können, sondern auch darum, wie diese im einzelnen implementiert sind — also um das, was gewöhnlich verheimlicht wird. Diese Verheimlichungen sind sicher berechtigt, aber da es sich bei HyperSpeed sozusagen um Public Domain Hardware handelt soll dem Benutzer alles offengelegt werden.

2.1 Speicher, Speicheraufteilung und was dazu gehört

2.1.1 RAM

Die Mindestkonfiguration beträgt wie schon erwähnt 128kB. Die Standardversion von HyperSpeed ist für drei 128kB RAMs ausgelegt. Der dritte RAM-Sockel kann auch mit 512kB S-RAMs bestückt werden. Dazu muß allerdings die MMU umprogrammiert werden.
Die RAMs teilen sich wie folgt auf:
RAM1 : \$000000 – \$01FFFF
RAM2 : \$020000 – \$03FFFF
RAM3 : \$040000 – \$05FFFF
Welcher RAM in welchen Sockel kommt, ist dem Placeplan zu entnehmen (Seite 80). Diese Nummerierung lässt sich auch durch umprogrammieren der MMU relativ einfach ändern.
Für Bank \$00 (also RAM1) gelten noch besondere Regeln (siehe ROM und ATARI Mapping).
Die Sockel für RAM1 und RAM2 sind zur Aufnahme von 128kx8 RAMs im 400mil breiten 32pol. SOJ Gehäuse in der Center Power Ausführung gedacht (siehe Abbildung 2.1). RAM3 kann nur RAMs im 32pol. 300mil oder 600mil breiten DIP Gehäuse in der herkömmliche JEDEC Ausführung aufnehmen. Bei diesem RAM gibt es noch etwas zu beachten. Und zwar muß durch den Jumper J3 eingestellt werden, ob es sich bei dem RAM um einen RAM mit zweitem High-aktiven ChipSelect handelt (J3: 2-3) oder nicht (J3: 1-2). Bei einem 128kB RAM sollte dies immer auf zweites ChipSelect eingestellt sein. Ein 512kB RAM darf nicht auf das zweite ChipSelect eingestellt werden, da sich bei diesem RAM an der Stelle eine Adressleitung befindet. Für den Chip hätte das zwar keine schlimmen Folgen, aber man hätte nur 256kB zur Verfügung. Der Jumper hat also nur die Aufgabe, entweder +5V (immer enabled) oder A18 bzw. BA2 an das betreffende Pin zu leiten. Die Chips müssen so eingesetzt werden, daß die Beschriftung auf diesen genauso wie die auf dem Placeplan zu lesen ist! Wird in den Sockel für RAM3 ein 300mil breiter Chip eingesetzt, dann wird dieser an Pin 16 (GND) ausgerichtet — also in die beiden unteren Reihen.

Abbildung 2.1: Pinbelegungen für CenterPower und JEDEC RAMs

128Kx8 32pin Center Power

Signal	Pin	Pin	Signal
A	1	32	A
A	2	31	A
A	3	30	A
A	4	29	A
/CS	5	28	/OE
DQ	6	27	DQ
DQ	7	26	DQ
Vcc	8	25	Vss
Vss	9	24	Vcc
DQ	10	23	DQ
DQ	11	22	DQ
/W	12	21	A
A	13	20	A
A	14	19	A
A	15	18	A
A	16	17	A

128Kx8 ... 512Kx8 32pin JEDEC

Signal	Pin	Pin	Signal
A	1	32	Vcc
A	2	31	A
A	3	30	A, CS2
A	4	29	/W
A	5	28	A
A	6	27	A
A	7	26	A
A	8	25	A
A	9	24	/OE
A	10	23	A
A	11	22	/CS
A	12	21	DQ
DQ	13	20	DQ
DQ	14	19	DQ
DQ	15	18	DQ
Vss	16	17	DQ

Adressleitungen (A) sowie Datenleitungen (DQ) sind hier jeweils voneinander ununterscheidbar dargestellt. Bei RAMs macht es im allg. keinen Sinn diese zu unterscheiden, weil Adress- und Datenabbildungen in jedem Fall eindeutig sind. Der Sockel für RAM3 ist aber so ausgelegt, daß dieser aufwärtskompatibel bis 512kB ist.

Warteschritte

Die Zugehörigkeit der Jumper zu den entsprechenden RAM Chips ist Tabelle 2.1 zu entnehmen. Steht der entsprechende Jumper auf 1-2, wird bei einem Zugriff ein ganzer Warteschritt ausgeführt, steht er auf 2-3 wird ein halber ausgeführt. Ist der Jumper nicht gesteckt, wird auch kein Warteschritt ausgelöst.

2.1.2 ROM

Der vorgesehene ROM Bereich beginnt mit Bank $F0 und endet mit Bank $FF (also 1MB). Sind nur 16kB ROM gesteckt, dann wiederholen sich diese 16kB 64 mal. Ob dieser große Bereich überhaupt notwendig ist, ist eine andere Frage. Wie auch immer, sollten bessere Lösungen auftauchen, so sind

Tabelle 2.1: Zuordnung der WaitState Jumper

Chip	Jumper
RAM1	J18
RAM2	J7
RAM3	J8
ROM	J9

Jumperlocation siehe Placeplan auf Seite 80!

Abbildung 2.2: Pinbelegungen für 28 und 32pol. ROMs bzw. 32pol. Flash

EPROM 64kx8 28pin JEDEC

Signal	Pin	Pin	Signal
A15	1	28	Vcc
A12	2	27	A14
A7	3	26	A13
A6	4	25	A8
A5	5	24	A9
A4	6	23	A11
A3	7	22	/OE
A2	8	21	A10
A1	9	20	/CS
A0	10	19	Q7
Q0	11	18	Q6
Q1	12	17	Q5
Q2	13	16	Q4
Vss	14	15	Q3

EPROM 1Mx8 32pin JEDEC

Signal	Pin	Pin	Signal
A19	1	32	Vcc
A16	2	31	A18
A15	3	30	A17
A12	4	29	A14
A7	5	28	A13
A6	6	27	A8
A5	7	26	A9
A4	8	25	A11
A3	9	24	/OE
A2	10	23	A10
A1	11	22	/CS
A0	12	21	Q7
Q0	13	20	Q6
Q1	14	19	Q5
Q2	15	18	Q4
Vss	16	17	Q3

5V Flash 512Kx8 32pin JEDEC

Signal	Pin	Pin	Signal
A18	1	32	Vcc
A16	2	31	/W
A15	3	30	A17
A12	4	29	A14
A7	5	28	A13
A6	6	27	A8
A5	7	26	A9
A4	8	25	A11
A3	9	24	/OE
A2	10	23	A10
A1	11	22	/CS
A0	12	21	DQ7
DQ0	13	20	DQ6
DQ1	14	19	DQ5
DQ2	15	18	DQ4
Vss	16	17	DQ3

Bei nicht aufgeführten Größen entfallen entsprechende Adressleitungen.

diese größtenteils durch Umprogrammieren der MMU realisierbar. Zurück zum Thema. Da sich nach einem Reset zumindest ein definierter Reset Vektor an entsprechender Stelle in Bank $00 befinden muß, muß sich an dieser Stelle auch ROM befinden. Es sei denn, man geht einen anderen Weg, indem man das *Vector Pull* Signal der 65C816 verwendet. Bei HyperSpeed V2.1 wird vorerst nur die ROM-Variante verwendet. Aus diesem Grund gibt es das Steuerbit *ROM_OS*, mit dem es möglich ist einen 16kB großen Block von $F0C000 – $F0FFFF in Bank $00 im Bereich von $00C000 – $00FFFF zu spiegeln. Ist *ROM_OS*=0, ist der ROM-Bereich unten eingeblendet. Mehr gibt es zum ROM im Bezug auf das Memory Mapping nicht zu sagen.

EPROM oder Flash?

Das Handling eines Flash's soll hier nicht erklärt werden. Mehr dazu befindet sich in der BIOS Dokumentation auf Seite **??**. Aufgrund minimaler Pinbelegungsunterschiede von EPROM und Flash sind aber ein paar Dinge zu beachten. Siehe dazu auch Abbildung 2.2. Bei der Verwendung von wahlweise 28poligen und 32poligen EPROMs gibt es keine größeren Probleme. Dazu muß nur mittels Jumper entweder A17 bzw. BA1 oder Vcc an Pin 30 des 32pol. Sockels gebracht werden. Das 28pol. EPROM kann dann rechtsbündig eingesteckt werden. Bei einem Flash muß insbesondere noch das */W* Signal mit angeschlossen werden. Außerdem gibt es noch eine Vertauschung einer Adressleitung, welche bei ROMs im Gegensatz zu RAMs beachtet werden sollte.
Wie die zur ROM-Auswahl notwendigen Jumper zu stecken sind ist Tabelle 2.2 zu entnehmen.

Warteschritte

Für die Warteschrittauswahl des ROMs ist J9 zuständig. Die Kodierungen befinden sich in Tabelle 2.2.

Tabelle 2.2: Einstellung des ROM Typ's und Warteschritte für ROM

J1	J16	J17	Typ
2-3	X	X	28Pin EPROM
1-2	2-3	1-2	32Pin EPROM
1-2	1-2	2-3	32Pin Flash

Kombination	Warteschritte
–	keine
1-4	0.5
1-3	1.0
1-2	1.5
1-5	2.0

Tabelle 2.3: Neue Memory Mapped Hardwareregister

Bereich	Verwendung
$EF00XX	MMU
$EF01XX	Real Time Clock
$EF02XX	VIA
$EF03XX	Single Step Manager
$EF04XX	reserviert
$EF05XX	allg. Nutzung
$EF06XX	allg. Nutzung
$EF07XX	allg. Nutzung

2.1.3 Hardwareregister

Im Adressraum von HyperSpeed existiert neben den ATARI Hardwareregistern ein zweiter Bereich für IO Hardware. Dafür ist eine komplette 64kB Bank (Bank $EF) reserviert. Es gibt eine primäre Leitung *New_IO*, welche den Zugriff auf den Bereich von $EF0000 – $EF1FFF signalisiert (Low aktiv). Dieses Signal steht auch für Erweiterungskarten zur Verfügung. Mit *New_IO* wird ein Decoder (74ACT138) angesteuert. Da der 74ACT138 noch zwei weitere Enables besitzt, werden diese Enables noch mit A11 und A12 angesteuert (A12 invertiert). Als Ergebnis erhält man acht Low aktive ChipSelects. Wie diese verwendet werden ist Tabelle 2.3 zu entnehmen. Die als 'reserviert' und 'allg. Nutzung' gekennzeichneten Signale werden dabei über die Slots nach außen gebracht.
Weiterhin werden die Hardwareregister des ATARIs ($00D000 – $00D7FF) noch einmal im Bereich von $EFD000 – $EFD7FF gespiegelt. Wozu das gut sein soll kommt noch.

2.1.4 ATARI Mapping

Der ATARI liegt prinzipiell in der nullten 64k-Bank, also im Bereich von $000000–$00FFFF. Wäre dieser Bereich nur für den ATARI festgelegt, hätte das einige unschöne Folgen:

- Da Stack- und Directbereich genau dort liegen, gibt es bei entsprechenden Operationen permanente Ausbremsungen der CPU. Auch dann, wenn der Code im neuen RAM liegen würde.
- HyperSpeed könnte nie ungestört für sich arbeiten, wenn die alte CPU läuft. Diese könnte ja zum Beispiel den Stackinhalt manipulieren, was fatale Folgen haben kann (sofern es nicht beabsichtigt ist). Ebenso wird die ATARI CPU von HyperSpeed gestört.

- Es wäre praktisch unmöglich, HyperSpeed von Anfang an mit einem seperaten BIOS zu booten, weil sämtliche Interruptvektoren und der Resetvektor auch in Bank $00 liegen.
- Ebenso wie das letztere ist der sperate Interruptbetrieb von HyperSpeed gefährdet. Daß heißt, die Vektoren für HyperSpeed könnten unbeabsichtigt vom ATARI geändert werden (z.B. ATARI-OS eingeschalten und die Vektoren sind mit sinnlosen Werten belegt).[1]

Alle diese Probleme würden sich also abschaffen lassen, wenn man den ATARI in Bank $00 einfach abschalten könnte und an dessen Stelle neuen Speicher einblendet. Dies ist aber auch nicht sonderlich flexibel. Sollen zum Beispiel bestehende Programme beschleunigt werden, dann müßte ja in Bank $00 der Komplette ATARI eingeblendet werden. Warum soll nun die CPU Speicherzugriffe (abgesehen von VIDEO RAM und Hardwareregistern) auf den langsamen ATARI-Speicher ausführen, wenn doch auch schneller RAM zur Verfügung steht? Also müßte es in Bank $00 Bereiche geben, an denen mal der ATARI und mal neuer Speicher liegt. Je feiner diese Bereiche sind, umso besser und effektiver läßt sich der 'virtuelle' ATARI Speicher zusammensetzen. Aber umso größer ist auch der kombinatorische Aufwand für die MMU. Aus diesem Grund ist Bank $00 nur in 5 Teilbänke zu je 16kB bzw. 8kB unterteilt. Für jede dieser 'kleinen' Bänke ist nun mit einem exclusiven Steuerbit wählbar, ob an der entsprechenden Stelle der ATARI oder neuer Speicher liegt.

Steuerbit	Adressbereich
ATARI_Mem_0	$000000 – $003FFF
ATARI_Mem_1	$004000 – $007FFF
ATARI_Mem_2_0	$008000 – $009FFF
ATARI_Mem_2_1	$00A000 – $00BFFF
ATARI_Mem_3	$00C000 – $00FFFF
ATARI_IO	$00D000 – $00D7FF

Ist ein Steuerbit 0, liegt an der entsprechenden Stelle neuer RAM bzw. ROM, ansonsten ATARI-Speicher (je nach dortiger Einstellung RAM oder ROM).
Mit einem weiteren Steuerbit *ATARI_IO* können noch zusätzlich unabhängig von *ATARI_Mem_3* die ATARI Hardwareregister ($00D000 – $00D7FF) ein- und ausgeblendet werden. Ist *ATARI_Mem_3* 1, ist dieser Bereich ohnehin eingeblendet (egal, wie *ATARI_IO* steht). Dies ist nützlich, da bei einem gewünschten ATARI-Hardwarezugriff seitens HyperSpeed nicht jedesmal die komplette 16KB Bank umgeblendet werden muß. Es ist in der Hinsicht auch notwendig, da im Falle eines Umschaltens auf den ATARI-Speicher ja auch die Interruptvektoren sozusagen flöten gehen und man müßte jedesmal aus Sicherheitsgründen jede potentielle Interruptquelle deaktivieren und hinterher wieder aktivieren. So ist alles 'on the fly' machbar.

Achtung: Wie schon erwähnt werden die Hardwareregister des ATARIs im Bereich von $EFD000 – $EFD7FF gespiegelt. Allerdings befindet sich dieser Bereich nur dann dort oben, wenn die Hardwareregister auch bei $00D000 – $00D7FF eingeblendet sind. Da die MMU schon bis oben hin zugepackt ist, hat sich das leider nicht anders in die Praxis umsetzen lassen. Große Nachteile dürften sich dadurch allerdings keine ergeben.

Jetzt kann also bei Standardsoftware zum Beispiel von $000000 – $007FFF neuer RAM und von $00C000 – $00FFFF kann auch neuer RAM (als ATARI OS Shadow) eingeblendet werden. Falls das

[1] Die 65C816 benutzt im Native-Mode andere Vektoren. Mehr dazu auf Seite 63.

Tabelle 2.4: Wahrheitstabelle für Bereich von $00C000 – $00FFFF

ATR_Mode	*ATARI_Mem_3*	*ROM_OS*	*ATR_OS*	Speicher
0	0	0	X	neuer ROM
0	0	1	X	neuer RAM
0	1	X	0	ATARI RAM
0	1	X	1	ATARI OS
1	0	0(X)	0	neuer RAM
1	0	0	1	neuer ROM
1	1	X	0	ATARI RAM
1	1	X	1	ATARI ROM

(Mit *ROM_OS* wird gesteuert, ob sich von $00C000 – $00FFFF neuer ROM ($ROM_OS = 0$) oder neuer RAM liegt.)

Tabelle 2.5: Steuerung der Hardware Protection Unit

HW_W_Protect	*HW_R_Protect*	Mode
0	0	disabled (default)
0	1	Exception only on Read
1	0	Exception only on Write
1	1	Exception on Read and Write

ATARI BASIC noch aktiv sein soll, kann dies auch noch im Bereich von $00A000 – $00BFFF shadowed werden. Der VIDEO RAM Bereich (unter Basic normalerweise von $008000 – $009FFF) kann nicht durch neuen RAM überblendet werden, weil ja so Schreibzugriffe nicht im VIDEO RAM landen würden. Legt nun das alte XL Programm unter diesen Umständen den VIDEO RAM dummerweise teilweise oder vollständig in einen Bereich, in dem neuer RAM liegt, dann gibt es Ärger. Dieser Ärger läßt sich aber mit einigen Klimmzügen umgehen (siehe ATARI Hardware Protection Unit Seite 16).
Ähnliche Probleme treten auf, wenn im ATARI andere Bankswitchings vorgenommen werden, wie RAM-Disk on/off, BASIC on/off und OS on/off. Diese Vorgänge lassen sich zwar auch mit der Hardware Protection Unit abfangen, aber bei sehr häufigen Umschaltvorgängen kann dies mehr Zeit kosten, als durch den schnellen Zugriff wieder hereingeholt werden kann. Aus diesem Grund gibt es einen sog. ATARI Mode. Dieser wird durch das Steuerbit *ATR_Mode* gesteuert. Befindet sich die MMU im ATARI Mode ($ATR_Mode = 1$) werden die Steuerleitungen, die die RAM-Disk (*ATR_RD*) und das ATARI OS (*ATR_OS*) ein- und ausschalten mit in die Zugriffssteuerung einbezogen. Wird nun auf den Bereich von $004000 – $007FFF zugegriffen und in diesem Bereich ist neuer RAM eingeblendet, aber die RAM-Disk (nach 130XE Standard) ist aktiv, dann wird der Zugriff auf den ATARI weitergeleitet. Mit dem ATARI OS Bereich wird in etwa das selbe gemacht. Befindet sich im Bereich von $00C000 – $00FFFF neuer ROM, aber das XL-Programm schaltet wie gewohnt über den PIA das OS aus, dann wird von $00C000 – $00FFFF neuer RAM eingeblentet. Man hat also noch schnellen Zugriff auf das RAM hinter dem OS.

2.1.5 ATARI Hardware Protection Unit

Wie schon beschrieben, handelt es sich hier um ein System mit welchem es möglich ist die Hardwareregister des ATARIs vor Zugriffen auf diese zu schützen. Dabei wird in Read- und Write-Zugriffe unterschieden. Die Steuerung erfolgt über die beiden Steuerbits *HW_W_Protect* und *HW_R_Protect*. Die entsprechenden Modi stehen in Tabelle 2.5. Tritt nun ein Zugriff auf den Bereich von \$00D000 – \$00D7FF auf (und an dieser Stelle liegt die ATARI Hardware), dann wird an der CPU ein Exception bzw. ein *ABORT* ausgelöst (siehe CPU Dokumentation). Dabei sind folgende Dinge zu beachten:

- *ABORT* darf nicht länger als einen CPU Zyklus aktiviert werden, weil sonst der ABORT auch abgebrochen wird.

- Das *ATARI*-Signal, welches angibt, ob auf den ATARI zugegriffen wird muß deaktiviert werden, weil ansonsten die DataBridge ein Write noch an den ATARI weiterreichen würde.

- Um nicht Gefahr zu laufen, daß die CPU durch einen falschen ABORT-Vector springt, wird automatisch neuer ROM im Bereich von \$00C000 – \$00FFFF eingeblendet. Dazu ist es notwendig, die Steuerbits *ATARI_Mem_3, ATR_Mode* und *ROM_OS* auf 0 zu setzen.

- In einem seperaten Bit *HPU_ABORT* wird gespeichert, daß der ABORT von der Hardware Protection Unit ausgelöst wurde. Damit kann die ABORT- Handling Routine den ABORT lokalisieren.

- Wird das Bit *HPU_ABORT* von der Service Routine ausgelesen, wird dieses gleich automatisch gelöscht. Das spart für die Routine Zeit...

Weiterhin darf nur ein ABORT ausgelöst werden, wenn *VDA*=1 ist (siehe CPU Docu). Damit wird vermieden, daß unter Umständen bei internen Operationen der CPU falsche ABORTs ausgelöst werden. Denn dann liegt mitunter eine ungültige Adresse auf dem Bus. So wird nur bei Datenzugriffen und eigentlich in diesem Bereich nie auftretenden OpCode Fetchs ein ABORT ausgelöst.
Speziell im Kontext der Hardware Protection Unit stehen auch die nach \$EFD000 – \$EFD7FF gespiegelten Hardwareregister des XL/XEs eine Rolle. Bei einem Zugriff auf diesen Bereich wird keine Exception ausgelöst. Das Betriebssystem verwendet diesen Bereich, wenn wirklich endgültig auf die ATARI-Hardware zugegriffen werden soll. Nun wird man sich fragen, wozu das gut sein soll. Denn das Betriebssystem könnte ja eigentlich auch den Schutzmechanismus aufheben, den Zugriff durchführen und hinterher den Schutzmechanismus wieder aktivieren. Dahinter steckt aber eine böse Falle, in die ich auch bald getreten wäre... Es kann ja hier nicht unterschieden werden, ob der Zugriff vom Betriebssystem oder einer Nutzerroutine kommt. Denn es gibt keine Prozeßklassen mit verschiedenen Privilegierungen (was einen echten Protected Mode ausmachen würde). Wird nun gerade nachdem der Schutz entfernt wurde ein Interrupt ausgelöst, dann läuft die Interruptroutine ohne Schutz. Also könnten dort keine Zugriffe auf die Hardwareregister abgefangen werden. Dies würde unter Umständen fatale Folgen nach sich ziehen. Softwaretechnisch könnte man das Problem umgehen, indem Interrupts grundsätzlich erst einmal in das Betriebssystem hinein erfolgen. Dort könnte man dann den Schutz aktivieren und zu entsprechenden Nutzerroutinen verzweigen. Da aber die Verbiegung von Interruptvektoren an der Tagesordnug ist, wäre das auch nicht die beste Lösung.
Bei eventuellen Weiterentwicklungen von HyperSpeed ist jetzt schon abzusehen, daß ein derartiger Schutzmechanismus nicht nur auf die ATARI-Hardware angewendet wird, sondern zumindest auch auf die restliche HyperSpeed Hardware.

2.1.6 Implementation der Memory Management Unit (MMU)

Die MMU steckt in einem ispLSI2032, d.h. sie kann wie schon erwähnt bei Bedarf direkt on Board umprogrammiert werden. Dies wird aber im Normalfall nicht notwendig sein.
Im Gegensatz zum XL/XE wird die MMU bei HyperSpeed direkt Memory Mapped gesteuert. Das bedeutet, daß hier nicht der Umweg über einen anderen Chip gegangen wird (beim XL/XE der PIA). Da die MMU nicht allzu teuer werden sollte und somit nicht sehr viele IO Leitungen zur Verfügung stehen (bei einem ispLSI2032 32), wird eine spezielle Eigenschaft der 65C816 CPU ausgenutzt. Diese Eigenschaft besteht darin, daß über den Datenbus der CPU gleichzeitig die oberen Adressleitungen — die Bankadresse — in die Außenwelt gebracht werden (mehr dazu in der CPU Dokumentation). Durch diese Tatsache werden einige Leitungen zur MMU eingespart. Schließlich müssen ja außer den Adressleitungen auch noch Datenleitunen zur MMU gebracht werden.

Bankadresse

Aus den multiplexten Bank/Datenleitungen muß nun innerhalb der MMU erst einmal die Bankadresse extrahiert — sprich gelatcht werden. Die einfachste Möglichkeit wäre, die Bankadresse mit Hilfe der D-FlipFlops des ispLSI2032 mit steigender Phi2 zu latchen bzw. registrieren. Das hat aber den Nachteil, daß die Bankadresse erst eine ganze Ecke (ca.5–7ns) nach der steigenden Phi2 zur Adressdekodierung zur Verfügung steht. Das hätte fatale Folgen für RAM Zugriffszeiten etc. Aus diesem Grund werden 'handgemachte' transparente Latches verwendet. Die Latches sind wie folgt aufgebaut:

```
BAx    = BAx.pin & Bank_L # BADx & !Bank_L;
Bank_L = Phi2_CPU # !RDY;
```

Ist *Bank_L* Low, wird der Eingang *BADx* auf *BAx* durchgeschalten. Geht *Bank_L* auf High, wird nicht mehr der Eingang durchgeschalten, sondern *BAx* selbst. *Bank_L* wird übrigens in der Praxis direkt substituiert. *Bank_L* ist High, wenn die Phi2 High ist, oder *RDY* LOW (CPU angehalten). Es ist sehr wichtig, daß *RDY* hier mit einbezogen wird, weil sonst im Falle *RDY*=0 falsche Bankadressen gelatcht würden (siehe CPU Documentation).

Write Daten speichern

Hierzu können im Gegensatz zum Bank-latchen die D-FlipFlops verwendet werden. Dabei werden die Daten nicht mit fallender *Phi2*, sondern mit steigender *Phi1* übernommen (siehe Clock Management). Das hat zum einen den Grund, daß bei dem verwendeten ispLSI2032 der externe Takt nicht invertiert an die internen FlipFlops gebracht werden kann, und somit die Daten nur mit steigendem Takt übernommen werden können. Der zweite Grund ist, daß durch die der *Phi2* etwas vorverlagerte *Phi1* intern im ispLSI die Daten etwa an der Stelle übernommen werden, an der auch die *Phi2* fällt. Somit werden Probleme mit Datenhaltezeiten vermieden. Es gibt ja schließlich im ispLSI auch Verzögerungen...
Die MMU besitzt derzeit 11 Bits, die auf zwei 8Bit Register aufgeteilt sind. Bis auf ein Bit sind alle Write only. Das Write-only rührt dabei nicht von funktionellen Eigenschaften her, sondern vom extremen Platzmangel im verwendeten ispLSI2032. Zum Ansteuern der Register werden also das niederwertigste Adressbit *A0*, die Read/Write Leitung *RW* und ein ChipSelect *CS* benötigt. Die Write-only Registerbits haben folgenden Aufbau:

```
Bitx_0.clk = Phi1;
Bitx_1.clk = Phi1;

Bitx_0.d = !CS & !A0 & !RW & BADx # (CS #  A0 # RW) & Bitx_0.q;
Bitx_1.d = !CS &  A0 & !RW & BADx # (CS # !A0 # RW) & Bitx_1.q;
```

Bitx_0 steht stellvertretend für alle Bits im nullten Register ($EF0000) und *Bitx_1* für alle im ersten Register ($EF0001). Die Gleichungen dürften weitestgehend selbsterklärend sein.
Bei den Steuerbits, die die Signale *ROM_OS, ATARI_Mem_3* und *ATR_Mode* repräsentieren, wird der ganze Ausdruck noch mit dem von der Hardware Protection Unit ausgelöstem (im ispLSI High-aktiven) *ABORT* invertiert AND-verknüpft. Ist das interne *ABORT*-Signal inaktiv, also Low, wirkt es sich auf das entsprechende Steuerbit nicht aus. Wird es aber aktiv, dann wird mit der fallenden *Phi2* bzw. mit der steigenden *Phi1* das Bit in den gewünschten Zustand automatisch umgeschalten. Mit dieser Flanke leitet ja auch die CPU den ABORT ein.
Das *HPU_ABORT* Bit, welches ja einen ABORT von der Hardware Protection anzeigt erfährt eine Sonderbehandlung. Dieses Bit ist ja das einzige Read-only Bit in der CPU. Es kann also nicht überschrieben werden. *HPU_ABORT* wird durch *Bit2_1* repräsentiert und hat folgenden Aufbau:

```
Bit2_1.clk = Phi1;

HPU_ABORT = (!Bit2_1.q & ABORT # Bit2_1.q) & !(!CS & A0 & RW);
Bit2_1.d = HPU_ABORT;
```

Mit dem ersten (geklammerten) Teil dieses Ausdruckes wird das Setzen des Bits im Falle eines ABORTs realisiert. Der zweite Teil (in der Klammer) ist ja nur High, wenn auf das erste Register der MMU lesend zugegriffen wird (Achtung: mit dem nullten geht's los). Da dieser Teil invertiert mit AND-verknüpft wird, hat das also zur Folge, daß unmittelbar nach einem Lesezugriff dieses Bit gelöscht wird. Da dies nicht innerhalb einer Nanosekunde geschieht, wirkt sich dieses Löschen aber nicht auf den Lesevorgang der CPU aus. Diese liest noch den alten Wert des Bits. Das ist sozusagen ein 'zerstörendes Lesen'.
Da also vom Bit2 auf dem Datenbus zur MMU auch gelesen werden soll, muß dieses Pin als Bidirektionales deklariert werden. Dazu wird ein Output Enable benötigt, welches wie folgt gebildet wird:

```
output.oe = Phi2_CPU & !CS & A0 & RW;
```

Der Ausgangstreiber ist also nur aktiv, wenn die *Phi2* High ist und lesend auf das erste Register der MMU zugegriffen wird.
Achtung: Die beiden Register wiederholen sich jeweils 128 mal im für die MMU reservierten Speicherbereich. Dies kann sich jedoch bei weiteren Entwicklungen sehr schnell ändern. Aus diesem Grund ist die Verwendung anderer Adressen als $EF0000 und $EF0001 philosophisch mit der Verwendung illegaler OpCodes gleichzustellen!!! Abgesehen davon sollten bei der Programmierung der MMU generell BIOS Routinen verwendet werden, es sei denn es kommt wirklich auf jeden Takt an.
Das ausführliche Listing der MMU befindet sich im Anhang.

2.2 Warteschrittmanager, HPhi2-Synchronizer, ISA-Interface

Diese drei Einheiten gehören funktionell zusammen, da HPhi2-Synchronizer und ISA-Interface den Warteschrittmanager (WSMan) unmittelbar für ihre Zwecke mit benutzen.

Tabelle 2.6: Codierung der noch zu bearbeitenden WS

W2	*W1*	*W0*	noch zu bearbeiten
0	0	0	0.0
0	0	1	0.5
0	1	1	1.0
0	1	0	1.5
1	1	0	2.0
1	1	1	2.5
1	0	1	3.0
1	0	0	3.5

Der WSMan wird derzeit extern über 4 OpenCollector Leitungen *WSTP05, WSPT10, WSTP15* und *WSTP20* angesteuert (Reihenfolge: 0.5WS – 2.0WS). Diese Leitungen werden im 1-aus-4-Code verwendet. D.h. ist eine entsprechende Leitung aktiv (Low), wird nur der eine Warteschritt ausgelöst. Sollten trotzdem zu einem Zeitpunkt mehrere dieser Leitungen aktiv sein, wird der WS abgearbeitet, der die meiste Zeit verbrät. Durch den 1-aus-4-Code können WS einfach (ohne zusätzliche Kombinatorik und mehrfach-Jumperung) bedient werden. Es wird praktisch nur das ChipSelect des betroffenen Chips benötigt. Dieses wird dann entweder mit einem OpenCollector Buffer oder mit einer Diode mit niedriger Flußspannung in ein OpenCollector Signal konvertiert und direkt auf die entsprechende WS-Leitung gelegt. Mit den OnBoard SpeicherChips wird dies auch getan. Das Prinzip des WSMan ist denkbar einfach. Zum dem Zeitpunkt, wenn normalerweise ein CPU-Zyklus beendet wird (mit fallender *Phi2*), wird ein Zähler entsprechend der auszuführenden WS geladen. Ist der Inhalt des Zählers null, sind also keine WS auszuführen. Andernfalls wird mit jeder steigenden Flanke des doppelten Taktes *DCLK* (siehe ClockManagement) der Zähler heruntergezählt. In der Praxis sind aber noch ein paar zusätzliche Dinge zu beachten.
Der Zähler ist 3 Bit breit. Demzufolge lassen sich 8 Warteschrittmöglichkeiten codieren: 0WS – 3.5WS. Die Standardversion von HyperSpeed unterstützt nur 0WS – 2.0WS. Sollte es aber aus irgendwelchen Gründen notwendig sein, daß längere WS benötigt werden, müsste lediglich der Zähler anders geladen werden. Dazu müsste das ispLSI1016, in dem sich der WSMan unter anderem befindet umprogrammiert werden. Die 3 Bits sind *W2, W1* und *W0*. Die Anzahl der noch abzuarbeitenden WS ist dort aus energiespar und elektrotechnischen Gründen im Grey-Code codiert (siehe Tabelle 2.6). Beim Grey-Code ändert sich beim Übergang von einem Zustand in einen benachbarten nur 1 Bit.

Hier sind erst einmal die booleschen Gleichungen des WSMan:

```
W3.clk=DCLK;
W2.clk=DCLK;
W1.clk=DCLK;
W0.clk=DCLK;

LoadCnt= !W3 & !W2 & !W1 & !W0 & !IO;

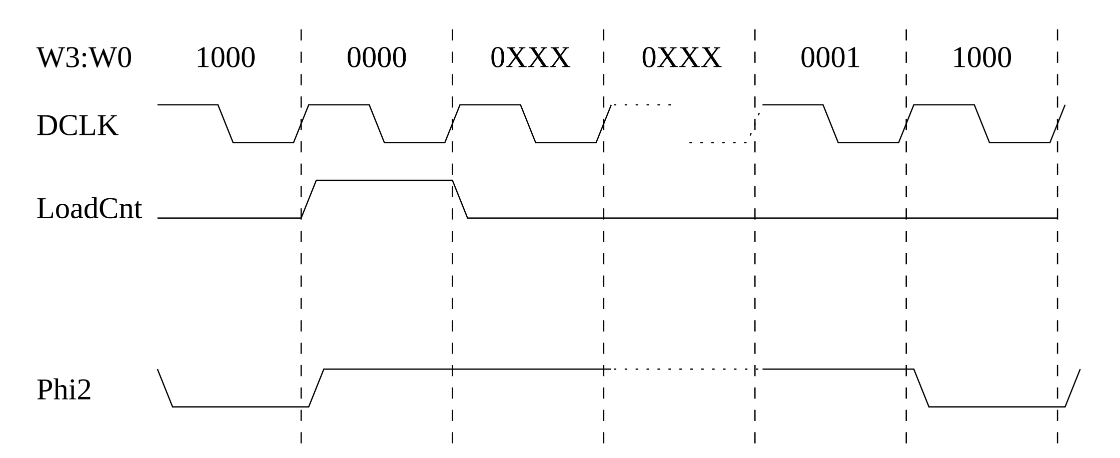
W3.d= !wait & (!W3 & !W2 & !W1 & W0 # !W3 & !W2 & !W1 & !W0 &
              (HPhi2Sel & WSTP20 & WSTP15 & WSTP10 & WSTP05) # IO));
W2.d= !LoadCnt & (W2 & W0 # W2 & !W1)
    #  LoadCnt & (HPhi2Sel & !WSTP20 # !HPhi2Sel & D4Q1);
W1.d= !LoadCnt & (W1 & !W0 # W2 & W1 # W2 & W0)
    #  LoadCnt & (!HPhi2Sel # HPhi2Sel & (!WSTP20 # WSTP20 & !WSTP15 # WSTP20
                                        & WSTP15 & !WSTP10));
W0.d= !LoadCnt & (!W2 & W1 # W2 & !W1)
    #  LoadCnt & (HPhi2Sel & (WSTP20 & WSTP15 & (WSTP10 & !WSTP05 # !WSTP10))
                 # !HPhi2Sel & D4Q0);
```

Das Signal (FlipFlop) *W3* stellt das Ausgangssignal des WSMan dar — nämlich den CPU-Takt bzw. die *Phi0* (siehe ClockManagement). Wenn man genau hinsieht, erkennt man, daß es sich bei *W3* um ein Toggle-FlipFlop mit mehreren Enables handelt. Als erstes gibt es das Signal *wait*. Ist dieses Signal 1, hat das ja zur Folge, daß *W3* immer auf 0 bleibt — die *Phi0* bzw. die *Phi2* an der CPU bleibt High. Das *wait*-Signal wird von der DataBridge generiert und wird im entsprechendem Abschnitt behandelt. Der nächste Teil `!W3 & !W2 & !W1 & W0` hat die Aufgabe eine WS-Sequenz zu beenden. Dieser Term geht auf 1, wenn noch genau ein halber WS abzuarbeiten ist. Also wird mit dem nächsten Takt der Wert 1 von *W3* übernommen, was zur Folge hat, daß die *Phi0* fällt und somit der CPU-Zyklus beendet wird. Weiter geht es mit `!W3 & !W2 & !W1 & !W0`. Hierbei handelt es sich um den Toggle-Teil im Normalfall. Ist kein WS am Laufen (*W2:W0*=000) wird mit jedem Takt der invertierte Wert von *W3* von *W3* übernommen, was ja offenbar einer Takthalbierung entspricht. Ein Umschalten von *W3* auf Low kann aber verhindert werden, wenn der letzte Term `((HPhi2Sel & WSTP20 & WSTP15 & WSTP10 & WSTP05) # IO)` Low ist. Dieser Term ist genau dann Low, wenn eine der WS-Leitungen aktiv (Low) ist, oder ein Zugriff über die HPhi2 erfolgen soll (HPhi2Sel Low). Sollte die CPU eine interne Operation durchführen (*IO* High) wird das ganze Theater deaktiviert, sodaß in diesem Fall keine WS ausgelöst werden. Mit *LoadCnt* wird festgelegt, wann der Zähler geladen wird (High). Ist *LoadCnt* Low wird der Zähler nicht geladen, sondern heruntergezählt (bei 0 bleibt er natürlich stehen). Bild 2.3 zeigt den Ablauf einer WS-Sequenz.

2.2.1 HPhi2-Synchronizer

Wie schon erwähnt handelt es sich bei der *HPhi2* im Gegensatz zur *Phi2* um einen exakt laufenden Takt. D.h. während die *Phi2* zum Beispiel durch das Einfügen von Warteschritten gestreckt wird, läuft die *HPhi2* mit einem konstanten Takt. Die *HPhi2* wird durch einen einfachen durch 4 Teiler (wieder im Grey-Code) aus dem doppelten Takt erzeugt:

Abbildung 2.3: Ablauf einer WS-Sequenz




```
D4Q1.clk=DCLK;
D4Q0.clk=DCLK;

D4Q1.d=D4Q0;
D4Q0.d=!D4Q1;
```

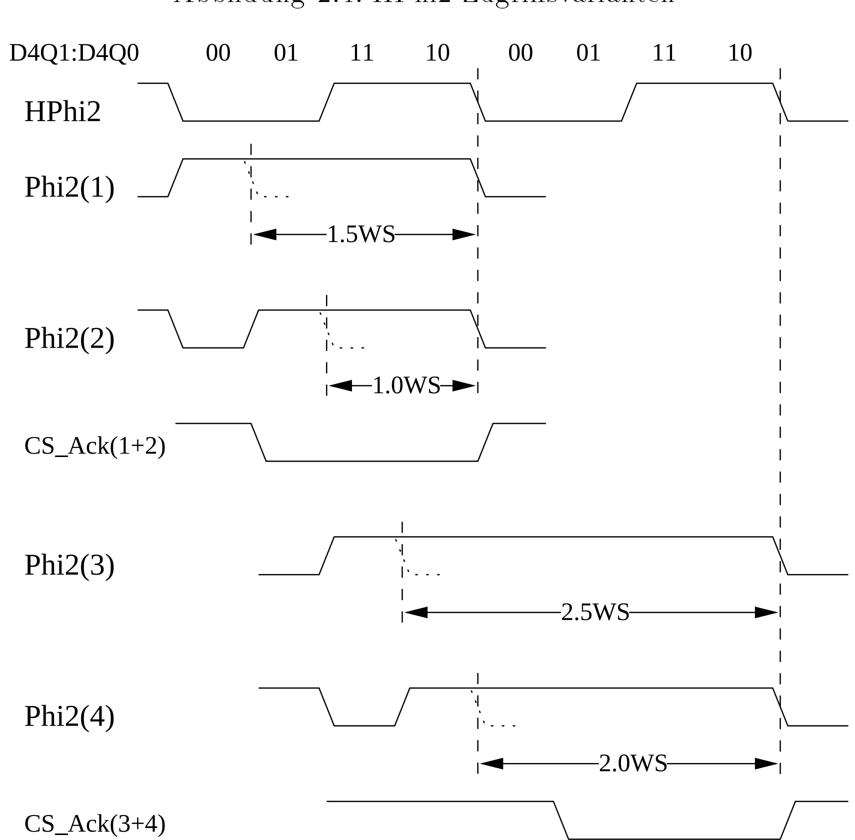
Das Signal *D4Q1* repräsentiert die *HPhi0*. Aus dem ispLSI wird diese aber vorerst invertiert (also als *HPhi1*) nach außen gebracht (siehe ClockManagement). Wird auf einen Chip zugegriffen, der zur *HPhi2* synchron läuft (z.Bsp. auf den OnBoard VIA), dann muß dies genau wie bei den Warteschritten über die *HPhi2Sel*-Leitung angemeldet werden. Dieses *HPhi2Sel* fließt in die Berechnung des Wertes ein, mit dem der WS-Counter geladen wird. Um das Timing der 6500er Serie einzuhalten, ist es nämlich je nach Zusammentreffen von *Phi2* und *HPhi2* notwendig 1.5, 1.0, 2.5 oder 2.0 Warteschritte einzulegen (Siehe Abbildung 2.4). Damit dürften die Gleichungen des WSMan vollständig beschrieben sein. Eine Betrachtung der Herleitung der Gleichungen zum Laden des WS-Counters möchte ich mir hier ersparen, da dies mehr oder weniger trivial ist...

Bezüglich des HPhi2-Synchronizers gibt es noch einen weiteren Punkt zu beachten: Das ChipSelect der betroffenen Chips. Dieses kann ja nicht mehr ohne weiteres asynchron an den Chip gebracht werden, weil dies zu Fehlern führen kann. Denn dadurch, daß ja im Normalfall die *Phi2* und die *HPhi2* praktisch asynchron laufen, kann es vorkommen, daß der HPhi2-Chip irgendwann einmal zu einem ungünstigen Zeitpunkt ein falsches ChipSelect (ein Spike am CS-Eingang) bekommt. Je nach Konstruktion des HPhi2-Chips könnte dies zur Folge haben, daß der Chip dann eine offenbar falsche Bus-Transaktion durchführt... Um diesen Mißstand zu beseitigen, wird vom HPhi2-Synchronizer ein Low-aktives ChipSelect Acknowledge zur Verfügung gestellt (*CS_Ack*).

```
CS_Ack= !HPhi2Sel & (
                         !W2 & (W1 # W0)
                       # !W3 & !W2 & !W1 & !W0 & !D4Q1 & D4Q0);
```

CS_Ack ist also wenn überhaupt , dann bei aktivem *HPhi2Sel* aktiv. *CS_Ack* wird aktiviert, wenn noch 1.5, 1.0 und 0.5 WS abzuarbeiten sind (`!W2 & (W1 # W0)`). Sollte der Fall mit Phi2(2) aus Abbildung 2.4 eintreten, wird *CS_Ack* schon in der ersten *Phi2*-High Phase aktiviert (Term `!W3 & !W2 & !W1 & !W0`

Abbildung 2.4: HPhi2-Zugriffsvarianten



```
& !D4Q1 & D4Q0).
```

Als Beispiel für die Anwendung des HPhi2-Synchronizers kann die Einbettung des OnBoard VIAs verwendet werden.

2.2.2 ISA-Interface

Hiermit werden die Leitungen *ISA_WR* und *ISA_RD* gesteuert. *ISA_WR* leitet beim ISA-Bus mit der fallenden Flanke eine Write-Operation und *ISA_RD* entsprechend eine Read-Operation ein. Nur gibt es bei den meisten asynchronen ISA-Chips (wie zum Beispiel beim OnBoard Realtime Chip DP8572) das Problem, daß bevor *ISA_WR* oder *ISA_RD* fallen eine strafbar lange Address Setup Time eingehalten werden muß. Beim DP8572 sind das zum Beispiel 20ns. Würde man also bei 14MHz mit der steigenden *Phi2* die ISA-Signale aktiv werden lassen, so würde diese Zeit weit unterschritten. Also dürfen diese Signale erst später aktiv werden. Bei HyperSeed ist wird dieses Problem ralativ einfach gelöst. Die ISA-Signale werden nämlich nur dann aktiviert, wenn Warteschritte abgearbeitet werden.

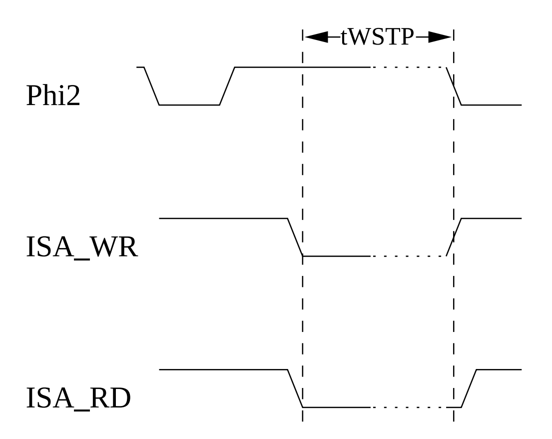
```
Pre_WR =!CPU_RW & (W2 # W1 # W0);
Pre_RD = CPU_RW & (W2 # W1 # W0);
```

Bei diesen Signalen handelt es sich noch nicht um die entgültigen ISA-Leitungen, sondern erst einmal um eine Vorstufe die im ispLSI generiert und nach außen gebracht wird. Die Signale sind hier High-aktiv, werden aber invertiert nach außen gebracht (siehe Listing). *Pre_WR* ist also nur aktiv, wenn der WS-Counter nicht auf null steht und die CPU schreibt. *Pre_RD* analog beim Lesen. Beim Einbinden eines ISA-Bausteins bzw. eines asynchronen Bausteins, der über *RD* und *WR* gesteuert wird, ist also praktisch nichts weiter zu beachten, außer der Tatsache, daß auf diesen Chip mindestens ein halber Warteschritt eingelegt werden muß.
Da mit der steigenden Flanke von *ISA_WR* der ISA-Chip die Daten auf dem Datenbus übernimmt, sollte diese Flanke also nicht irgendwann, sondern zu einem möglichst günstigen Zeitpunkt auftreten. Aus diesem Grund bildet sich das *ISA_WR* aus der Veroderung von *Pre_WR* und der *Phi1*. Also tritt die steigende Flanke an *ISA_WR* genau dann auf, wenn die *Phi2* an der CPU fällt. WDC garantiert eine Data Hold Time von 10ns. Das sollte normalerweise genügend sein. Der Realtime Chip DP8572 z.Bsp. verlangt nach der steigenden Flanke des Write-Signals 3ns Hold Time. Auf der anderen Seite ist mir auch ein Chip bekannt (AD1848KP — 16Bit Stereo Sound Codec von Analog Devices), der laut Datenblatt 25ns Data Hold Time verlangt. Hier wäre es besser *ISA_WR* einen halben Warteschritt eher zu deaktivieren. Dann kommen bei 14MHz noch ca. 35ns hinzu. In diesem Fall müßte demzufolge mindestens ein Warteschritt ausgelöst werden. Da früher oder später evtl. doch generell diese Variante der Write-Steuerung verwendet werden könnte, sollte auch bei der jetzigen Version mindestens ein ganzer WS auf die betroffenen Bauteile eingelegt werden (es sei denn, die WS-Anzahl läßt sich flexibel einstellen). Damit würden Probleme gleich von vornherein ausgeschlossen. Außerdem muß beachtet werden, daß die minimale *ISA_WR* Low Time des Bausteins etwa um 35ns überschritten wird. Denn diese Zeit würde ja später evtl. abgezogen.
Das *Pre_RD* kann eigentlich schon so verwendet werden. Wie auch immer, es wird aber noch mit einem OR-Gatter gepuffert.
Abbildung 2.5 zeigt noch einmal den zeitlichen Verlauf der ISA-Signale.

2.3 ATARI RealTime Data Bridge

Vom Prinzip her ist die Implementation der DateBridge recht einfach. Es gibt zwei Steuerungen — eine auf der HyperSpeed-Seite und eine auf der ATARI-Seite, einen Buffer zum zwischenspeichern der

Abbildung 2.5: ISA-Timing

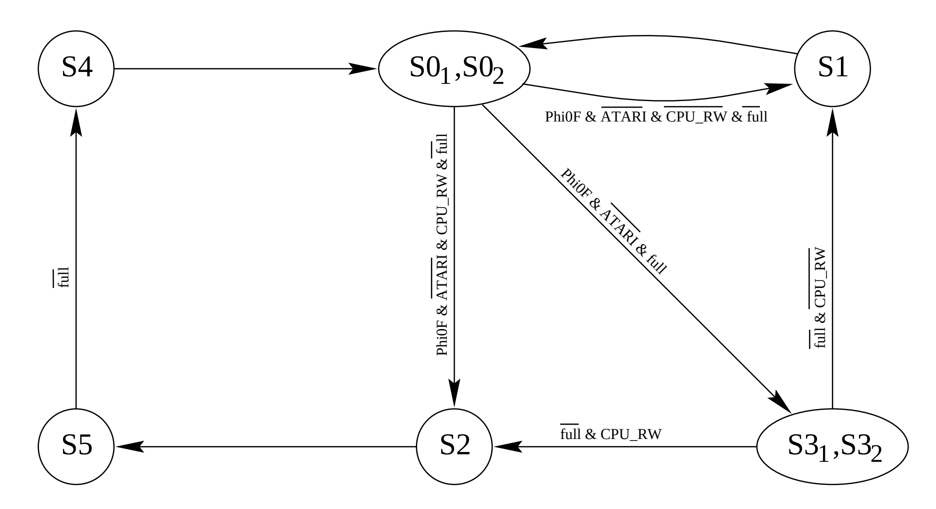


Adresse und Daten und eine Semaphore die angibt, ob der Buffer voll oder leer ist. Das hört sich so recht einfach an, ist es in der Praxis aber nicht. Denn hier kommen einige erschwerende Faktoren hinzu. Zum einen ist da das vollkommen asynchrone Verhalten zwischen HyperSpeed und dem ATARI. Denn HyperSpeed soll ja mit einem X-beliebigen, vom ATARI unabhängigen Takt betrieben werden. Als nächstes soll die DataBridge eine minimale ATARI-Request Latency bringen. Soll heißen, daß HyperSpeed in Echtzeit mit dem ATARI kommunizieren muß, um wenigstens die Pace des normalen XL/XEs mitgehen zu können. In der Realität ist HyperSpeed aber noch schneller als das Original, auch wenn ausschließlich auf dem ATARI-Speicher operiert wird. Dazu aber später mehr.
Die Bridge wie sie jetzt vorliegt dürfte noch nicht optimal sein, aber viel mehr wird mit diesem Prinzip nicht herauszuholen sein. Es sei denn, man führt einen mehrstufigen Write-Buffer oder einen Cache ein. Das führt dann aber wieder zu anderen Problemen.

2.3.1 HyperSpeed-Seite (V3.2)

Abbildung 2.6 zeigt den (prinzipiellen) HyperSpeed-seitigen Automaten der DataBridge. Er wird mit dem Systemtakt (*DClk*) getaktet. Dieser Automat arbeitet allerdings nicht nach Schema 'F', wie es allgemein üblich ist. Hier soll nicht die gesamte Automatentheorie durchgekaut werden, auf jeden Fall habe ich hier meine eigene Strategie entwickelt, um Automaten zu entwickeln die zuverlässig mit asynchronen Signalen umgehen können. Geht man nämlich den normalen Weg, dann passiert es ganz schnell, daß ein asynchrones Eingangssignal in den Überführungsfunktionen mehrerer FlipFlops auftaucht. Dadurch sind Fehler schon vorprogrammiert. Legt man zum Beispiel an zwei einfache D-FlipFlops eine asynchrone (asynchron zum Takt der FlipFlops) Eingangsgröße an, dann kann nicht garantiert werden, daß beide FlipFlops jeweils den selben Wert übernehmen. Denn die FlipFlops sind technisch bedingt niemals so identisch, daß sie sich in jedem Fall gleich verhalten, wenn der Takt kommt und sich gleichzeitig der Eingang ändert.
Bei HyperSpeed stellt die *full*-Semaphore die asynchrone Eingangsgröße dar. Denn diese wird von der ATARI-Seite der DataBridge asynchron zum HyperSpeed-Takt gelöscht. Eine einfache Lösung des Problems wäre die Synchronisation von *full* mit einem FlipFlop etwa mit dem fallenden doppelten Takt. Bei einem HyperSpeed Systemtakt von 28MHz (also 14MHz CPU) hätte das allerdings zur Folge, daß sich die Reaktionszeit des Automaten um ca. 18ns verschlechtert. Das klingt jetzt vielleicht

Abbildung 2.6: DataBridge - HyperSpeed seitiger Automat



nicht so tragisch, aber wenn sich *full* kurz (z.B. 2-3ns) nach der fallenden Flanke von *DClk* ändern würde, dann reagiert der Automat ja nicht auf diese Änderung, weil ja im synchronisations-FlipFlop noch der alte Wert gespeichert ist. Also wird die Änderung erst mit dem nächsten Takt des Automaten registriert — 36ns Verlust. Das hört sich auch noch nicht sehr viel an, aber falls gerade 'haarscharf' ein ATARI-Zyklus verpaßt wurde, dann schlägt sich das beim Lesen in diesem speziellen Fall gleich mit einer 'Strafzeit' von ca. 560ns nieder. Das entspricht 8 Taktzyklen der 65C816 bei 14MHz. Sicher treten gerade solche Fälle nicht oft auf, aber je träger die gesamte Steuerung ist, umso wahrscheinlicher werden diese Fälle. Wie soll nun dieses 18ns-Loch beseitigt werden, ohne dabei ein fehlerhaftes Design zu bekommen? Ganz einfach: *full* darf nur an ein FlipFlop des Automaten gebracht werden. Das entspricht dann sozusagen einer OnLine-Synchronisation. Das erfordert aber auch ein entsprechendes Handling beim Entwurf eines solchen Automaten.
Zur Bedeutung der Zustände:

S0

S0 ist der Idle-Zustand. Dieser Zustand ist wie angedeutet in der Realität die Vereinigungsmenge zweier Zustände. Denn es gibt ja die zwei Möglichkeiten, daß die *full*-Semaphore entweder 0 oder 1 ist.

S1

Hier wird eine Write-Operation auf den ATARI bzw. auf den Zwischenbuffer ausgeführt. Im einzelnen bedeutet das, daß die unteren 16Bit der HyperSpeed-Adresse und die 8Bit-Daten auf dem Datenbus von HyperSpeed in dem Zwischenbuffer gespeichert werden. Gleichzeitig muß die *full*-Semaphore gesetzt werden, um der ATARI-Seite der DataBridge mitzuteilen, daß die Daten im Buffer abholbereit sind.

S2

In diesem Zustand wird eine Read-Operation seitens HyperSpeed eingeleitet. Es werden wieder die niederwertigen 16Bit der Adresse im Adressregister des Zwischenbuffers gespeichert und *full* wird gesetzt. Außerdem wird hier der CPU-Takt auf High gehalten, denn leider muß ja jetzt im Gegensatz zum Schreiben gewartet werden, bis das entsprechende Datum aus dem XL/XE gelesen wurde. Das Latchen der Adresse ist eigentlich nicht nötig, da ja ohnehin die Adresse über den gesamten Read-Vorgang konstant auf dem HyperSpeed Bus anliegt. Aber da für den Adressbuffer reine Register (keine transparenten) verwendet werden, muß ja die Adresse irgendwie dort hinein gebracht werden.

S3

Das ist der Zustand, in den man nicht gelangen sollte ;-). Dort hinein gerät der Automat nämlich dann, wenn ein Zugriff auf den ATARI erfolgt, obwohl der Buffer noch voll ist (also läuft noch ein alter Write-to-ATARI Vorgang). Hier tritt wieder so ein State-Aliasing wie bei S0 auf, denn beim Übergang von S0 in S3 wird nicht unterschieden, ob gelesen oder geschrieben wird. Hier wird nichts weiter gemacht, als den CPU-Takt auf High zu halten. Wird dann irgendwann der Buffer einmal leer, dann wird je nach dem R/W Signal der CPU in S1 oder S2 übergegangen.

S5

Hier wird praktisch die gesamte Zeit 'totgeschlagen', die die ATARI-seitige DataBridge benötigt, um das Datum aus dem ATARI zu lesen. Also wird hier solange geblieben, bis *full* Low ist und der CPU-Takt auf High gehalten. Durch *full*=Low signalisiert nämlich die ATARI-Seite, daß das entsprechende Byte aus dem ATARI gelesen wurde.
Theoretisch ist dieser Zustand nicht notwendig und könnte eigentlich mit Zustand S2 zusammengefaßt werden, aber erstens stört dieser Zustand nicht (er bremst die Bridge nur, wenn der HyperSpeed-Takt extrem kleiner als der ATARI-Takt ist) und zweitens ergibt er sich mehr oder weniger aus dem physischen Entwurf des Automaten.

S4

Hier wird ein Read-from-ATARI abgeschlossen. Also muß spätestens hier der Datenbuffer auf den HyperSpeed-Bus geschalten werden. Während sich der Automat in diesem Zustand befindet, bleibt die *Phi2* an der CPU den letzten *DClk*-Zyklus auf High und geht auf Low, wenn in den Zustand S0 übergegangen wird.
Dieser Zustand ist mir auch noch ein Dorn im Auge. Denn durch diesen verlängert sich ein Read-form-ATARI Zyklus eigentlich unnötigerweise um den einen *DClk*-Zyklus. Es könnte ja zum Beispiel der Datenbuffer schon während des gesamten Zustandes S5 auf den HyperSpeed-Bus geschalten werden und wenn *full* auf Low geht könnte mit dem nächsten Takt (*DClk*) sofort in S0 gewechselt werden und die *Phi2* müsste auch sofort fallen. Es müsste nur noch gewährleistet sein, daß das gelesene Byte schon etwa 10ns im Buffer steht (wegen Data Setup Time der 65C816) — dies dürfte aber schon wegen der relativ langsamen *full*-Semaphore automatisch der Fall sein. Leider ist mir bis jetzt noch nichts eingefallen, wie man das unter Beachtung des asynchronen *full*-Signals bewerkstelligen könnte. Auf jeden Fall dürfte es auf eine Verschmelzung von WaitState Manager und DataBridge (HyperSpeed-Seite) hinauslaufen. Derzeit besteht ja eher eine lose Verbindung, indem durch das *wait*-Signal signalisiert wird, daß die Phi2 auf High bleiben soll.

Steuersignale

Die Steuersignale für die Überführungen zwischen den Zuständen dürften klar sein. Zum einen gibt es die schon desöfteren erwähnte *full*-Semaphore (wenn *full*=1 ist der Buffer voll).
Das *ATARI*-Signal kommt direkt von der MMU und gibt mit Low an, daß ein ATARI-Zugriff vorliegt. Im Automatengraphen ist es zwar nicht mit aufgeführt, aber das *ATARI*-Signal wird nur akzeptiert, wenn die CPU keine internen Operationen ausführt. Damit wird verhindert, daß evtl. ungültige Adressen einen falschen ATARI-Zugriff auslösen.
Phi0F ist der der *Phi2* etwas vorverlagerte Takt (siehe auch WaitState Manager und Clock Management). Die *Phi0F* wird benötigt, daß der Zustand S0 nur am Ende eines normalen CPU-Zyklus verlassen wird und nicht schon in der Phase wo die *Phi2* bzw. *Phi0F* erst auf High geht. Denn dort kann noch nicht gewährleistet werden, daß das *ATARI*-Signal schon gültig ist (insbesondere bei hohem CPU-Takt). Würde die CPU nur mit 10MHz betrieben, dann könnte ein ATARI-Zugriff bedenkenlos schon dort begonnen werden. Aber HyperSpeed ist für den maximalen Takt optimiert...
CPU_RW ist einfach nur das *R/W*-Signal des HyperSpeed Busses bzw. der 65C816.

Überführungsfunktionen

Die booleschen Gleichungen des Automaten sind verblüffend einfach. Es gibt vier FlipFlops *FDB[3:0]* zum Speichern des Zustandes. Eine rationale Erklärung, wie man auf die Gleichungen kommt kann ich leider nicht angeben. Es handelt sich eher um eine Random-Logik, die mehr durch scharfes Denken als durch ein striktes Vorgehen entsteht.

```
FDB3.clk=DCLK1;
FDB2.clk=DCLK1;
FDB1.clk=DCLK1;
FDB0.clk=DCLK1;
Phi0F= !W3; // Substitution von Phi0F (siehe auch WSMan)
//Substitution der Zustandscodierungen (uebersichtshalber)
S1    = FDB3 & !FDB2 & !FDB1 & !FDB0;
S2    = FDB3 &  FDB2 & !FDB1 & !FDB0;
S5    = FDB3 &  FDB2 &  FDB1 &  FDB0
S4    = FDB3 &  FDB2 & !FDB1 &  FDB0;

FDB3.d= Phi0F & !ATARI & !IO & !S1 & !S4;
FDB2.d= Phi0F & CPU_RW & !ATARI & !IO & !S1 & !S4;
FDB1.d= full;
FDB0.d= S2 # S5;
```

Tabelle 2.7 zeigt die Codierung der einzelnen Zustände. *FDB3* ist eigentlich nur ein D-FlipFlop, was das *ATARI*-Signal (invertiert) speichert. In allen Zuständen außer S0 ist *FDB3* High. *FDB3* kann also nur auf High gehen, wenn die *Phi0F* auf High, *ATARI* auf low ist und keine interne Operation ausgeführt wird (*IO* Low). Außerdem wird in den Zuständen S1 und S4 erzwungen, daß mit dem nächsten Takt *FDB3* auf Low geht um den Übergang in S0 zu erreichen.
Mit *FDB2* geschieht prinzipiell dasselbe, nur wird hier der Zustand der *R/W*-Leitung gespeichert.
FDB1 macht nichts weiter, als einfach nur das *full*-Semaphor mit jedem Takt neu zu übernehmen.
FDB0 ist ein FlipFlop im Sinne der herkömmlichen Automatentheorie. Dies wird benötigt, um die

Tabelle 2.7: DataBridge - HyperSpeed Automat Zustandcodierung

Zustand	FDB[3:0]
S01	0000
S02	0010
S1	1000
S2	1100
S31	1010
S32	1110
S4	1101
S5	1111

Zustände zu erfassen (S5 und S4), die nicht allein durch die anderen FlipFlops unterschieden werden können.

Ausgangsfunktionen

Aus S0 heraus muß der CPU-Takt auf High gehalten werden, wenn vom ATARI gelesen werden soll oder wenn bei einem allg. ATARI-Zugriff der Buffer voll ist. Ansonsten würde ja in all den Fällen die *Phi0* schon auf Low gehen. Außerdem muß die *Phi0* in den Zuständen S3, S2 und S5 auf High bleiben (Wartezustände). Diese ganze Anhalterei geschieht über das High-aktive *wait*-Signal, welches wie folgt gebildet wird:

```
wait =   FDB2 & !FDB1 & !FDB0 # FDB3 & FDB1            //S2 # S3 # S5
       # !FDB3 & !FDB2 &  FDB1 & !FDB0 & !ATARI & !IO   //Buffer voll (S0_2)
       # !FDB3 & !FDB2 & !FDB0 & !ATARI & CPU_RW & !IO; //Lesen
```

Anmerkung: In ungünstigen Fällen kann es ja bei Write-to-ATARI Operationen vorkommen, daß *FDB1* noch High ist, während *full* schon auf Low ist (weil eben *full* erst kurz nach dem Speichern in *FDB1* auf Low gegangen ist). In diesem Fall wird trotzdem das *wait*-Signal aktiviert (und somit die *Phi0F* für den nächsten *DClk*-Zyklus auf High gehalten), da ja der HyperSpeed-Automat noch nichts von der Änderung von *full* erfahren hat. Diese Tatsache ist aber nicht weiter schlimm und hat nur den Effekt, daß bei manchen (eben solchen) Schreibvorgängen auf den ATARI praktisch ein halber Warteschritt eingefügt wird. Diese Situation tritt aber in vielleicht 1% aller Fälle auf und dürfte nicht weiter ins Gewicht fallen. Außerdem lässt sich das gar nicht vermeiden, denn durch das asynchrone Verhalten von *full* muß irgendwann ein Schlußstrich gezogen werden...

In den Zuständen S1 und S2 muß die niederwertige 16Bit Adresse vom HyperSpeed Bus in den Adressbuffer übernommen werden. In S1 muß zusätzlich noch das Byte vom Datenbus in den Datenbuffer übernommen werden. Dafür sind folgende (High-aktiven) Signale zuständig:

```
latch_H_ADDR = FDB3 & !FDB1 & !FDB0;
latch_H_Data = FDB3 & !FDB2 & !FDB1;
```

Gleichzeitig mit dem Latchen der Adresse wird noch der aktuelle Wert der *RW*-Leitung der 65C816 im Register *A_RW* gelatcht. Dieser Wert gehört ja eigentlich mit zur Adresse. Anhand von *A_RW* wird dann entweder ein Write- oder ein Read-Zyklus auf den ATARI getrieben.

```
A_RW.ptclk=latch_H_ADDR;
A_RW.d=CPU_RW;
```

Im Zustand S4 muß das vom ATARI gelesene Byte, das jetzt im Datenbuffer steht auf den HyperSpeed Datenbus gelegt werden. Dazu gibt es das *H_Data_EN*-Signal (High-aktiv):

```
H_Data_EN = !FDB1 & FDB0;
```

Anmerkung: Da *H_Data_EN* so relativ früh wieder inaktiv wird, was zu Problemen bzgl. der Data Hold Time der 65C816 führen könnte, wird dieses Signal noch einmal durch das ispLSI1016 geschleift...

Zum Schluß wird noch das Signal zum Setzten der *full*-Semaphore benötigt. Dieses Signal ist identisch mit dem zum Latchen der Adresse.

```
F_full_set=FDB3 & !FDB1 & !FDB0;
```

Einige Timingverläufe

Abbildung 2.7 zeigt einen Ablauf, in dem zweimal unmittelbar hintereinander auf den ATARI geschrieben wird. Vor dem ersten Write ist der Buffer leer. Hier ist auch noch deutlich zu sehen, daß bei dem zweiten Schreibvorgang der CPU-Takt *Phi2* bzw. *Phi0F* im Zustand S1 noch auf High ist (im Gegensatz zum ersten Write). Das läßt sich aber wie schon gesagt nicht vermeiden. Ansonsten riskiert man Fehler.
Die Signale *latch_H_ADDR* sowie *latch_H_Data* heißen dort übrigens nur *l_H_ADDR* bzw. *l_H_Data* (weil das andere so lang ist ;-).
In Abbildung 2.8 ist zu sehen wie ein einzelner Read-Vorgang der 65C816 auf den ATARI abläuft. Hier ist der Buffer zu Beginn auch leer.
Bei Abbildung 2.9 erfolgt auch ein Read auf den ATARI. Hier ist aber der Buffer zu Beginn noch voll — es läuft also noch ein alter Schreibvorgang. Der Timingverlauf ist dort nicht vollständig angegeben, da praktisch ab dem Zustand S2 alles genauso abläuft wie bei Abbildung 2.8.

2.3.2 ATARI-Seite

Hier wird erst einmal der ATARI-Takt in 8 einzelne Phasen zerlegt. Dazu ist der **Phi0S - Phase Dedector** (*PPD*) zuständig. Diese Einheit wird mit der steigenden Flanke des 8-fachen ATARI-Taktes (hier *EACLK* genannt) getaktet. Bei den neueren ATARI XE-Modellen wird dieser 8-fache Takt auf dem Motherboard erzeugt und im DRAM-Control-Chip (FREDDY / CO61991) durch vier geteilt. Bei den XLs ohne FREDDY wird leider nur der doppelte CPU-Takt erzeugt. Aus diesem Grund wird dort noch eine kleine Zusatzplatine benötigt, welche den 8-fachen Takt erzeugt, diesen an HyperSpeed weiterleitet und gleichzeitig durch 4 geteilt anstelle des alten doppelten XL-Taktes in das XL-System einspeist. Dabei muß dort die Phasenlage zwischen *Phi0S* und *EACLK* möglichst genauso wie beim XE hingebogen werden. Die Gleichungen des PPD lauten wie folgt:

Abbildung 2.7: Zwei aufeinanderfolgende Writes auf den ATARI

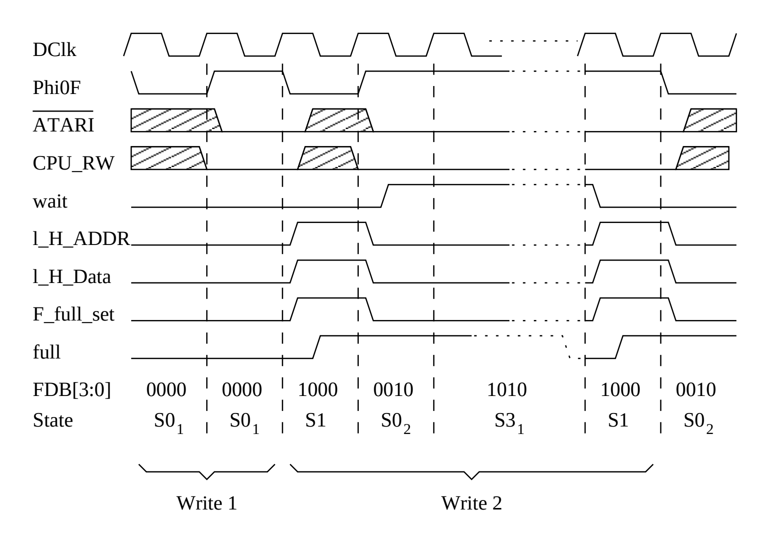


Abbildung 2.8: Ein Read auf den ATARI

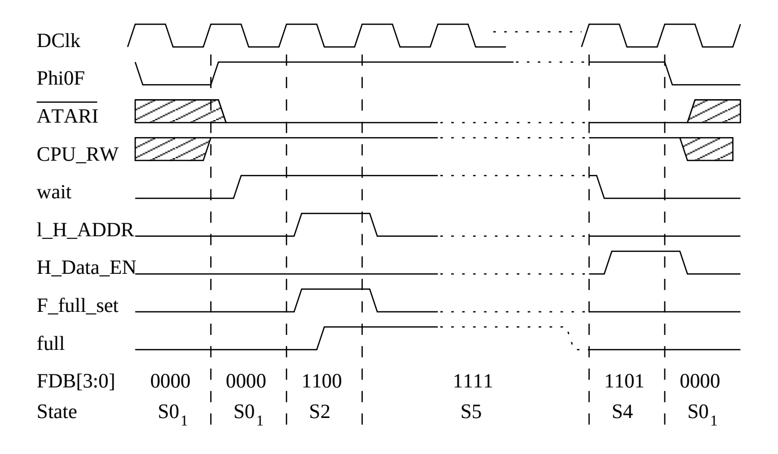
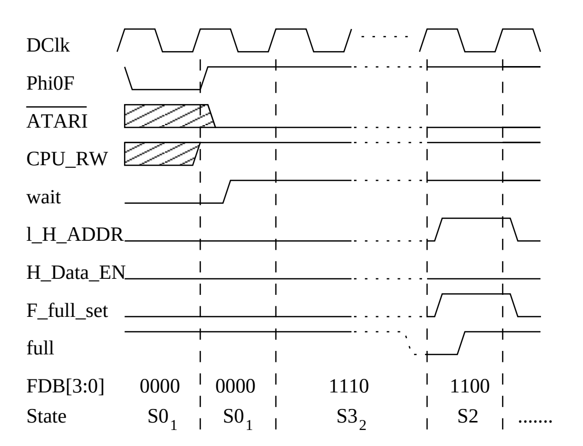


Abbildung 2.9: Ein Read auf den ATARI bei noch vollem Buffer



Nach S2 geht es genauso weiter, wie bei Abb. 2.8 nach S2.

```
PDQ2.clk=EACLK;
PDQ1.clk=EACLK;
PDQ0.clk=EACLK;

PDQ0.d=  !PDQ2 & !PDQ1 & !PhiOS
       #  PDQ2 &  PDQ1 &  PhiOS;

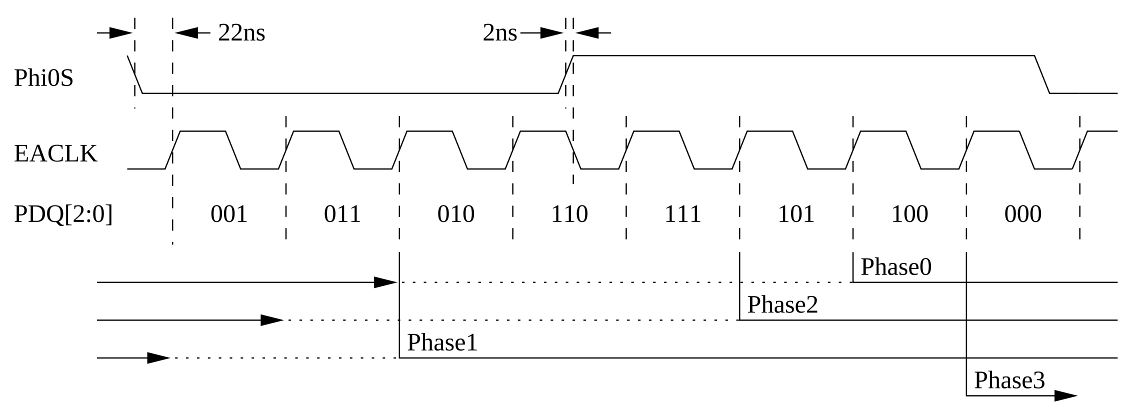
PDQ1.d=  !PDQ2 & PDQ0 & !PhiOS
       # !PDQ2 & PDQ1 & !PDQ0 & !PhiOS
       #  PDQ2 & PDQ1 & !PDQ0 &  PhiOS;

PDQ2.d=  !PDQ2 & PDQ1 & !PDQ0 & !PhiOS
       #  PDQ2 & PDQ1 & !PDQ0 & PhiOS
       #  PDQ2 & PDQ0 & PhiOS;
```

Der PPD zählt die 8 Phasen im Crey-Code durch und hat die wichtige Eigenschaft, daß er sich nach dem Einschalten des Systems automatisch phasenrichtig zur *Phi0S* (das ist der Phi0-Takt des XL/XEs) kalibriert. Ohne die Kalibrierung würde der PPD irgendwo zufällig mit zählen beginnen was natürlich nicht Sinn und Zweck der Sache ist. Der PPD 'geht' spätestens nach dem ersten *Phi0S*-Zyklus nachdem das globale *Reset* vom ATARI wieder inaktiv wurde richtig, so daß es da keine Probleme gibt. Abbildung 2.10 zeigt den zeitlichen Verlauf von *EACLK, Phi0S* und die Kodierung der einzelnen Phasen. Was dort die einzelnen Phasen zu bedeuten haben folgt weiter unten.

Was wurde nun mit dieser Aufteilung erreicht? Durch die feinere Teilung wird zum einen die Wahrscheinlichkeit erhöht, daß ein gerade laufender ATARI-Taktzyklus noch dazu verwendet werden kann, eine Bustransaktion seitens HyperSpeed auszuführen. Würde zum Beispiel nur die *Phi0S* bzw. *Phi2S* verwendet, dann muß mit der fallenden Flanke dieses Taktes festgestellt werden, ob eine Bustransaktion durchgeführt werden muß (also ob der Buffer voll ist). Wenn ja, dann wird im nun folgendem Takt ein ganz normaler 6500er Buszyklus ausgeführt — also mit der fallenden Flanke wird die Adresse auf den Bus gelegt und mit der steigenden Flanke im Write-Fall die Daten. Stellt man sich nun vor,

Abbildung 2.10: Die Phasen des Phi0S Phase Dedectors



Phi0S_Low = 270ns / Phi0S_High=290ns (etwa)

EACLK_Low =30ns / EACLK_High=40ns (etwa)

daß der Buffer aber erst kurz (paar ns) nach der fallenden Flanke des ATARI-Taktes von HyperSpeed gefüllt wird, dann kann ja der gerade angefangene Zyklus nicht verwendet werden und es muß gewartet werden, bis der komplette laufende ATARI-Zyklus vorbei ist. Bei der Original 6502 CPU haben meine Messungen ergeben, daß dort die Adresse erst etwa 50-70ns vor der steigenden Flanke auf dem Bus stabil ist. Also bleibt dort ein Zeitraum von mindestens 200ns der noch abgewartet werden könnte bis entschieden wird, ob ein Buszyklus auf den XL/XE getrieben werden muß. Weiterhin wird die feinere Teilung mehr oder weniger benötigt, um das Timing auf den ATRI-Bus feiner zu gestalten. Bei der veralteten Technik werden zum Beispiel mit hoher Wahrscheinlichkeit relativ hohe Hold-Times (after Write) benötigt, so daß die Bustreiber aus Sicherheitsgründen nicht einfach unmittelbar nachdem die *Phi0S* wieder auf Low geht abgeschalten werden können. Genausowenig darf die Adresse nicht unmittelbar nach der fallenden Flanke der *Phi0S* auf den ATARI-Bus geschalten werden, weil es dort unter Garantie Konflikte mit gerade auslaufenden Buszyklen des ANTIC oder der alten CPU gibt (die Treiber werden aufeinander geschalten). Als zusätzliches Problem kommt noch hinzu, daß der XL/XE als BlackBox betrachtet werden muß, da praktisch nichts über bestimmte Details bekannt ist. So kann man zum Beispiel nur durch experimentieren herausbekommen, was die minimale Address Setup Time (before *Phi0S* High) ist. Zu allem Übel wird sich sicher jeder XL/XE in dieser Hinsicht noch wie ein eigenes Individuum verhalten...

Es sind aber noch andere Dinge zu beachten. Nun arbeiten ja drei Einheiten auf dem ATARI-Bus: Die alte CPU, der ANTIC und HyperSpeed. Der ANTIC benötigt aus bekannten Gründen nach wie vor die höchste Priorität. Aus diesem Grund muß die *Halt*-Leitung des ATARIs ausgewertet werden. Die Original XL/XE-CPU latcht die *Halt*-Leitung mit der fallenden *Phi0* (das ist ausnahmsweise bekannt). Ist an dieser Stelle *Halt* Low, dann bedeutet das, daß der sich unmittelbar anschließende Zyklus für den ANTIC reserviert ist. HyperSpeed latcht also prinzipiell auch an dieser Stelle das *Halt*-Signal und verfährt entsprechend.

Ist die alte CPU nicht über das *Hlt_A*-Signal (siehe ATARI Control) fest angehalten, dann ist auch Vorsicht geboten. Es gibt ein Signal *Halt_Atari*, das in das *Halt*-Signal der alten CPU mit eingemischt wird. Über dieses Signal kann die DataBridge die alte CPU anhalten um einen Buszyklus auf den ATARI zu treiben. Hier (wenn die alte CPU läuft) kann aber nun nicht im letzten Moment mit dem

Buszyklus begonnen werden. Denn die alte CPU registriert ja eine Änderung an ihrem *Halt*-Eingang nur mit der fallenden Flanke der *Phi0*. Aus diesem Grund muß hier sichergestellt werden, daß das *Halt_Atari*-Signal mindestens eine gewisse (unbekannte) Setup Time (before Phi0S Low) aktiv ist. Ist dies gewährleistet, dann kann der folgende ATARI-Zyklus von HyperSpeed verwendet werden. Das bremst die Bridge natürlich wahnsinnig aus, aber das soll auch nicht der Normalzustand sein. Die alte CPU soll eigentlich nur dazu da sein, um bei Bedarf auch noch alte zeitabhängige Programme oder solche, die illegale OpCodes nutzen laufen lassen zu können. Trotzdem ist es auch möglich beide CPUs parallel arbeiten zu lassen. Es ist aber bei HyperSpeed 2.1 nichts vorgesehen um etwa echtes Multi Processing oder dergleichen effektiv zu unterstützen. Mehr dazu bei ATARI Control.
Nun aber zu den einzelnen Phasen. Ist die alte CPU nicht über deren *Halt*-Leitung angehalten, dann muß Phase0 im Vorfeld eines ATARI-Buszyklus seitens HyperSpeed mindestens einmal durchlaufen werden. Über diesen Zeitraum hinweg wird auch *Halt_Atari* aktiviert. Länger braucht *Halt_Atari* nicht aktiviert werden. Würde dies über den gesamten Zyklus aktiv gehalten, dann hätte das nur den Nachteil, daß die alte CPU unnötigerweise auch noch den nachfolgenden Zyklus angehalten wird. Phase1 stellt den Zeitraum dar, in dem die Adresse auf den Bus geschalten wird und Phase2 den für die Write-Daten. Mit Eintritt in Phase3 werden beim Lesen die Daten vom ATARI in den Buffer übernommen. Das mag zwar etwas früh erscheinen, aber die alte CPU braucht ja auch im Gegensatz zur heutigen Technik eine relativ lange Data Setup Time. Messungen haben auch ergeben, daß der Datenbus schon lange bevor die *Phi0S* fällt stabil ist. Gleichzeitig mit dem Latchen der Daten wird die Semaphore gelöscht, um dem HyperSpeed-seitigen Automaten mitzuteilen, daß das Byte gelesen wurde. Mit dem Ende von Phase3 kann beim Schreiben die Semaphore gelöscht werden.

Anmerkung: Für die einzelnen Phasen werden später keine extra Signale im ispLSI eingeführt, sondern es werden immer die entsprechenden Ausdrücke direkt substituiert.

Steuerung des ATARI-Buszugriffes

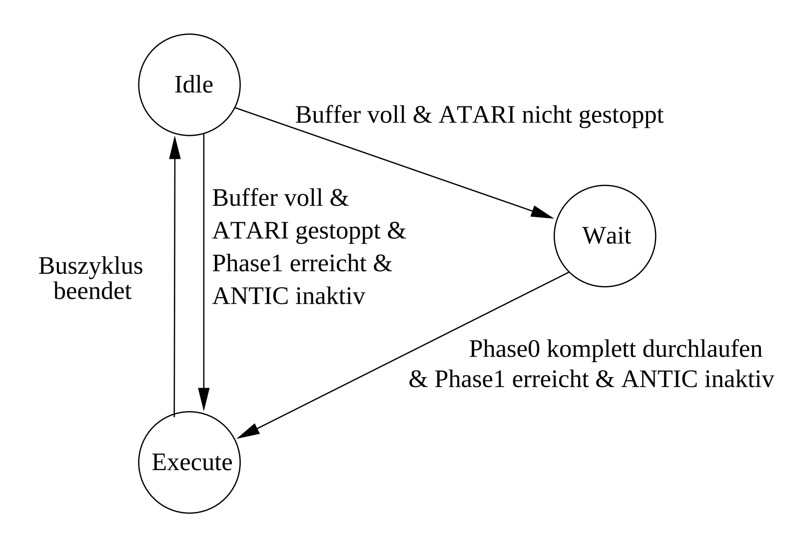
Abbildung 2.11 zeigt noch einmal verbal ausgedrückt den prinzipiellen Ablauf eines Zugriffs auf den ATARI. Die Zustände dort haben aber nicht notwendigerweise etwas mit der konkreten Realisierung zu tun.
Die die Gleichungen des ATARI seitigen DataBridge-Automaten:

```
SDB_full.clk   =!EACLK;
SDB_Phase0.clk =!EACLK;
SDB_Phase1.clk =!EACLK;

//Substitution der einzelnen Phasen
//PhaseX_b = Beginn von PhaseX
Phase0_b =  PDQ2 & !PDQ1 & !PDQ0;
Phase0   = !PDQ1 & !PDQ0 # !PDQ2 & PDQ0;
Phase1_b = !PDQ2 & PDQ1 & !PDQ0;
Phase1   = !PDQ0 # PDQ2;

SDB_full.d   = full;
```

Abbildung 2.11: Prinzipieller Ablauf eines ATARI-Zugriffs




```
SDB_Phase0.d = iA_Hlt &
               ( !Hlt_A  //ATARI Bus Sequenz kann sofort begonnen werden
                # Hlt_A &
                  (!SDB_Phase1 & !SDB_Phase0 & Phase0_b #
                    SDB_Phase0 & Phase0 & SDB_full
                  )
               );
SDB_Phase1.d = !SDB_Phase1 & Phase1_b & SDB_Phase0
              # SDB_Phase1 & SDB_full & Phase1;
```

Der Automat (wenn man das so bezeichnen kann) wird mit der fallenden Flanke von *EACLK* getaktet. *SDB_full* übernimmt wieder die Realtime-Synchronisation der *full*-Semaphore. *SDB_Phase0* fungiert als eine Art Enable-Bit, daß *SDB_Phase1* auf 1 gehen kann. Wie *SDB_Phase0* gebildet wird dazu später mehr. *SDB_Phase1* wird während *Phase1_b* gesetzt, aber nur unter der Bedingung, daß *SDB_Phase0* High ist. Damit wird ein Buszyklus auf den ATARI eingeleitet. Zu beachten ist hier, daß zu diesem Zeitpunkt noch kein Unterschied gemacht wird, ob der Buffer voll oder leer ist. Mit dem zweiten Term wird *SDB_Phase1* über die gesamte *Phase1* auf High gehalten. Allerdings nur dann, wenn der Buffer auch wirklich voll ist. Andernfalls wird *SDB_Phase1* mit dem nächsten Takt wieder auf 0 gesetzt. Praktisch wird also schon mal ein ATARI-Buszyklus eingeleitet, in der Hoffnung, daß auch einer nötig ist. Dabei werden die Adressleitungen auch nur auf den ATARI-Bus gelegt, wenn der Buffer voll ist. Da das Latchen der Semaphore und das Einleiten des Buszyklus parallel geschehen, kann somit sozusagen in letzter Nanosekunde noch ein evtl. gerade eintreffendes Request seitens HyperSpeed mit dem laufenden ATARI-Zyklus abgefertigt werden. Allerdings nur dann, wenn *SDB_Phase0* auf 1 ist.
SDB_Phase0 kann wenn überhaupt, dann wenn *iA_Hlt* High ist auf 1 gehen. *iA_Hlt* ist die schon vorbehandelte *Halt*-Leitung vom XL/XE bzw. vom ANTIC. Wie schon beschrieben wird ja der folgende Buszyklus vom ANTIC verwendet, wenn zur fallenden *Phi0S* diese *Halt*-Leitung Low ist. Ist also *iA_Hlt* Low, dann ist *SDB_Phase0* auch Low und somit kann *SDB_Phase1* nicht auf High gehen und es kann von HyperSpeed kein Buszyklus auf den ATARI getrieben werden. Ist *iA_Hlt* auf High, dann muß

nun entschieden werden, ob die alte ATARI-CPU fest angehalten wurde (*Hlt_A* ist Low; siehe auch bei ATARI Control). Mit dem Beginn von *Phase0* wird nun *SDB_Phase0* gesetzt, falls die alte CPU nicht angehalten wurde. *SDB_Phase0* bleibt die gesamte *Phase0* auf High, wenn wieder parallel zum Setzen von *SDB_Phase0* auch *SDB_full* gesetzt wurde bzw. schon gesetzt ist. Während *SDB_Phase0* und *SDB_full* auf High sind, wird die *Halt*-Leitung der alten CPU auf Low gesetzt. Zu beachten ist hier, daß *SDB_Phase0* nur gezetzt wird, wenn *SDB_Phase1* Low ist (also die DataBridge keinen alten Buszyklus mehr abarbeitet). Andernfalls gäbe es zwar keinen Fehler, aber dadurch, daß zu diesem Zeitpunkt der Buffer ja noch voll ist, würde fälschlicherweise erkannt, daß im nächsten ATARI-Zyklus eine Bustransaktion durchzuführen wäre. Demzufolge würde also *SDB_Phase0* erst mal gesetzt. Spätestens drei Takte (fallende Flanke von *EACLK*) später wird aber *SDB_Phase0* wieder gelöscht, da ja mit dem Ende des alten Buszyklus die Semaphore gelöscht wird. Es sei denn, die HyperSpeed-Seite war so schnell, daß die Semaphore schon wieder gesetzt werden konnte. In diesem Fall hätte das sogar den Vorteil, daß der aktuelle ATARI-Zyklus gleich von HyperSpeed genutzt werden kann und erhöht somit den mittleren Datendurchsatz. Allerdings hätte das wiederum den Nachteil, daß die alte CPU grundsätzlich bei jedem Zugriff zwei ATARI-Takte angehalten wird — einmal für den Zyklus in dem die Bus-Transaktion durchgeführt wird und für den darauffolgenden Zyklus. Da die alte CPU schon ohnehin nicht sehr schnell ist, wird sie also noch mehr unnötig gebremst. Hinzu kommt noch die Gefahr, daß unter Umständen die alte CPU extrem lange nicht mehr zum Zuge kommt und deren Registerinhalte verloren gehen können.

Ausgänge der ATARI-Seite

Die Adresse wird mit dem Signal *H_ADDR_EN* auf den ATARI-Adressbus geschalten. *H_ADDR_EN* ist intern im ispLSI High-aktiv, aber extern Low-aktiv.

```
A_ADDR_EN=SDB_Phase1 & SDB_full;
```

Die Adresse wird also nur durchgeschalten, wenn Phase1 läuft und der Buffer voll ist.
Mit der Adresse muß auch die *RW*-Leitung auf den ATARI geschalten werden. Der der Wert von *RW* nicht wie die Adresse in einem externen Register gespeichert werden, wird die Tri-State Funktion des ispLSIs verwendet.

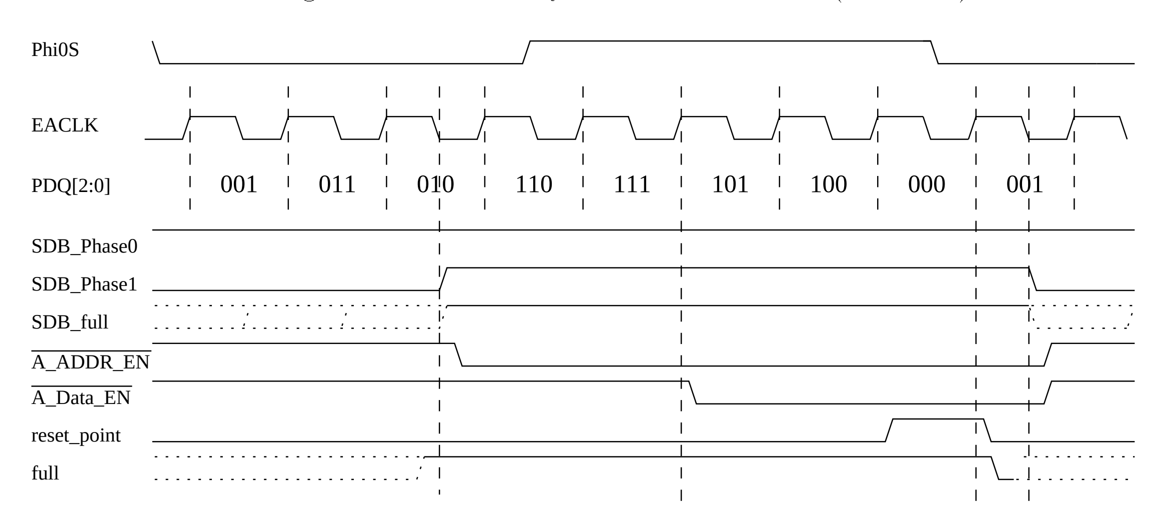
```
RW_EN.oe=A_ADDR_EN;
ATARI_RW=A_RW;
```

Im Falle einer Schreiboperation wird über die *Phase2* hinweg der Datenbuffer auf den Datenbus des XL/XEs gelegt.

```
Phase2 = !PDQ2 & !PDQ1 & PDQ0 # !PDQ1 & !PDQ0 # PDQ2 & !PDQ1;
A_Data_EN=!A_RW & Phase2 & SDB_Phase1;
```

Zu beachten ist hier, daß die Daten hier nicht die gesamte *Phase2* durchgeschalten werden. Den letzten *EACLK*-Low Zyklus ist *A_Data_EN* nich mehr aktiv, da ja *SDB_Phase1* mit der fallenden Flanke von *EACLK* bei *PDQ[2:0]*=001 auf Low geht. Die Data Hold Time dürfte aber trotzdem noch lang genug sein. Unter Umständen ist es auch möglich, daß von der HyperSpeed-Seite schon neue Daten in den Buffer übernommen werden, noch bevor der Daten-Buffer vom XL/XE abgekoppelt wurde. Das macht aber nichts, da das Timing so gestaltet ist, daß die Write-Daten schon übernommen wurden sein sollten. Mehr dazu weiter unten...
A_Data_EN ist extern Low-aktiv.
Wird vom ATARI gelesen, dann werden die Daten mit dem Eintritt in *Phase3* in den Daten-Zwischenbuffer übernommen. Dies geschieht über das High-aktive Signal *latch_A_Data*.

Abbildung 2.12: Ein Write-Zyklus auf den ATARI (*A_RW*=0)




```
Phase3_b = !PDQ2 & !PDQ1 & !PDQ0;
latch_A_Data = A_RW & Phase3_b & SDB_Phase1;
```

Sollte die alte CPU im XL/XE nicht fest über deren *Halt*-Leitung angehalten worden sein, dann muß das bei einem Zugriff auf den ATARI automatisch geschehen.

```
Halt_Atari = SDB_Phase0 & SDB_full;
```

Halt_Atari ist extern Low-aktiv.
Weiterhin gibt es noch ein Signal zum Rücksetzten der Semaphore, um dem HyperSpeed-Automaten je nach Situation entweder mitzuteilen, ob der Datenbuffer beim Lesen gefüllt oder beim Schreiben geleert wurde. Wie das Rücksetzen funktioniert soll im nächsten Abschnitt beschrieben werden.

Timingverläufe

Um die Funktion der ATARI-seitigen DataBridge noch einmal zu verdeutlichen, folgen auch wieder ein paar Timing-Diagramme.
Abbildung 2.12 zeigt den Verlauf der wichtigsten Signale bei einem Write. Dabei wird davon ausgegangen, daß die alte CPU über deren *Halt*-Leitung angehalten wurde und der ANTIC keinen Buszyklus durchführt.
Bei Abbildung 2.13 ist der Verlauf eines Read-Zyklus zu sehen (Bedingungen wie oben).

2.3.3 Aneinanderbindung der beiden DataBridge-Seiten

In Abbildung 2.14 sind die einzelnen Komponenten der DataBridge sowie deren Ein- und Ausgänge dargestellt.

Abbildung 2.13: Ein Read-Zyklus auf den ATARI (A_RW=1)

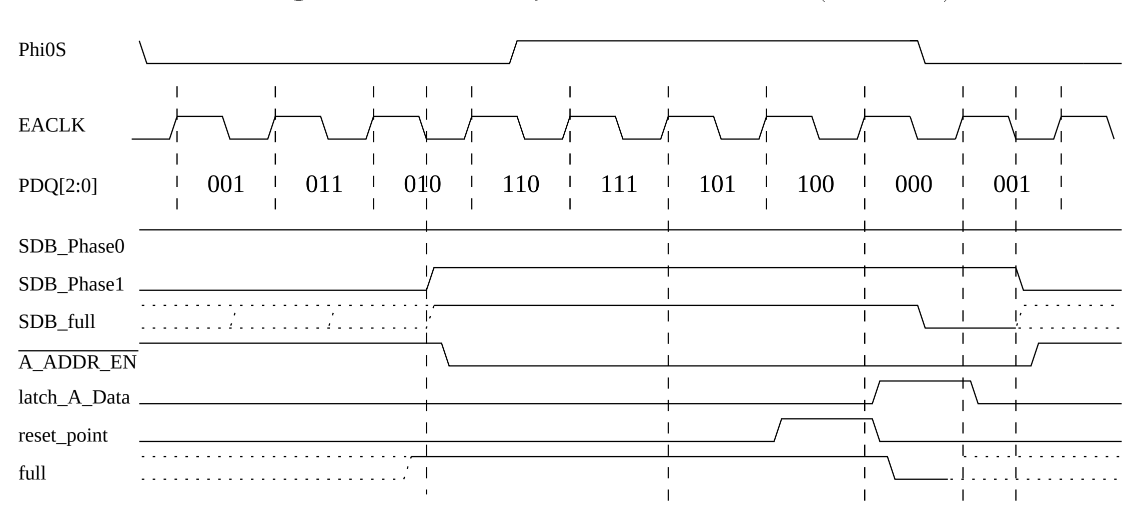
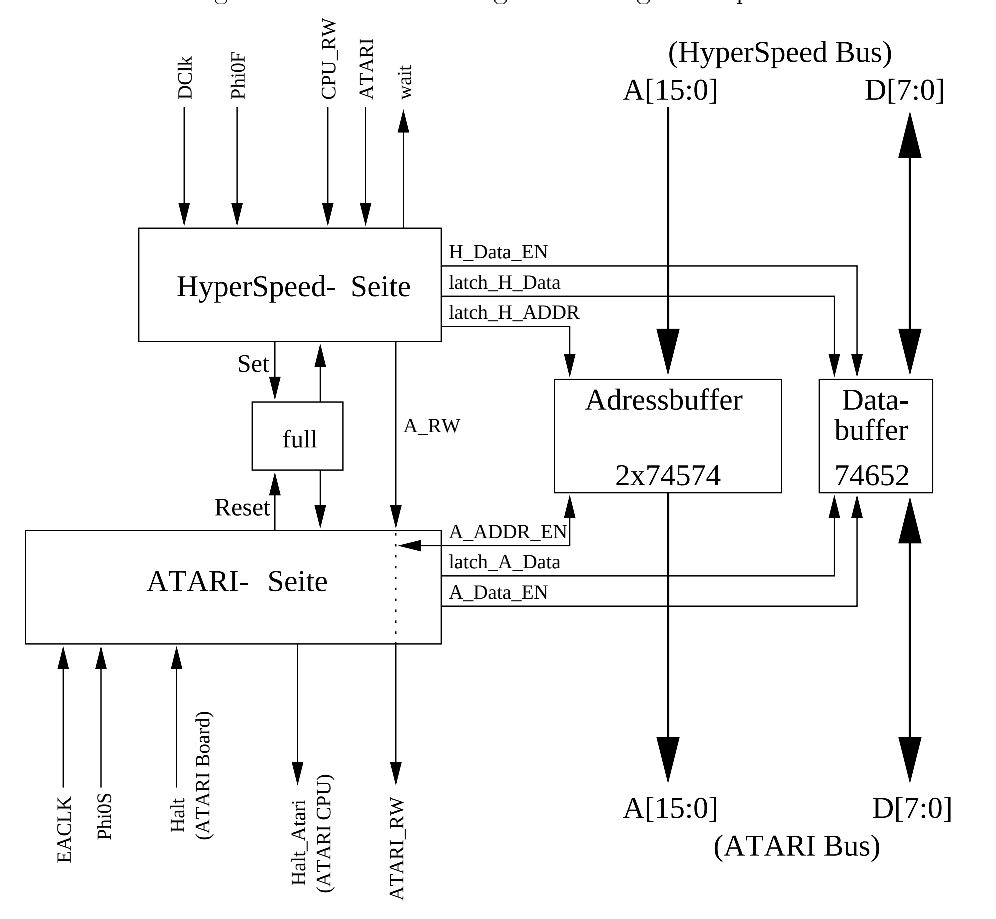


Abbildung 2.14: Zusammenhänge der Bridge-Komponenten



Datenpfad

Zum Zwischenspeichern der 16Bit Adresse werden zwei D-Register mit jeweils 8Bit verwendet (74ABT574). Mit der steigenden Flanke am Takteingang (*latch_H_ADDR*) übernehmen diese Register die für den ATARI relevanten Adressbits A[0:15]. Mit einem Low-aktiven Enable-Eingang (*A_ADDR_EN*) werden deren Ausgänge getrieben und somit auf den ATARI-Bus geschalten.
Für den Datenbuffer wird ein etwas komplizierterer Baustein verwendet. Der 74ABT652 ist ein 8Bit Bustransceiver mit je einem 8Bit Register an jedem Port. Die Funktionen die dieser Baustein bietet sind schon recht vielfältig. Hier soll nur kurz das beschrieben werden, was bei HyperSpeed genutzt wird. Mit einer steigenden Flanke an *CLKAB* (Pin 1) wird das Byte gespeichert, welches zu diesem Zeitpunkt an Port A anliegt. Im konkreten Fall ist das bei HyperSpeed das Signal *latch_A_Data*, mit welchem das vom ATARI gelesene Byte gespeichert wird. Analog dazu wird mit der steigenden Flanke an *CLKBA* (Pin 23) der Wert an Port B gespeichert (entspricht *latch_H_Data*). Mit dem High-aktiven Eingang *OEAB* (Pin 3) wird das in Port A gespeicherte Datum auf Port B ausgegeben (*H_Data_EN*). Umgekehrt erfolgt die Ausgabe des in Port B gespeicherten Datums auf Port A über den Low-aktiven Eingang *OEBA* (Pin 21) bzw. *A_Data_EN*.

full-Semaphore

Vom Prinzip her müßte *full* ein asynchrones RS-FlipFlop sein — mit einer Leitung wird full gesetzt und mit einer zweiten rückgesetzt. Optimal wäre es noch, wenn die Setz- und Rücksetzeingänge nur auf Flanken reagieren würden. Denn wenn die Takte der beiden Seiten (XL/XE und HyperSpeed) sehr unterschiedlich sind, dann könnte beispielsweise folgender Fall eintreten:
Die langsamere Seite aktiviert die Reset-Leitung von *full* und *full* geht kurz darauf auf 0. Die schnellere (meinetwegen unendlich schnell) Seite registriert dies und aktiviert ihrerseits die Set-Leitung von *full*. Da die langsame Seite gegenüber der schnellen sozusagen unendlich träge ist, hält sie die Reset-Leitung immer noch aktiv und tut dies auch noch lange nachdem die schnelle Seite die Set-Leitung schon wieder deaktiviert hat. Es geht also der Set-Vorgang der schnellen Seite verloren.
Bei einem HyperSpeed Systemtakt von 28MHz kann dieser Fall durchaus mit relativ hoher Wahrscheinlichkeit eintreten, wenn zum Rücksetzten ein voller *EACLK*-Zyklus (70ns) und zum Setzen ein voller *DClk*-Zyklus (35ns) benötigt wird. Wie das möglich ist kann ja einmal als eine Art Übungsaufgabe zum besseren Verständnis der DataBridge nachvollzogen werden...
Nun gibt es im ispLSI1016 weder RS-FlipFlops geschweige denn RS-FlipFlops mit den geforderten Eigenschaften. RS-FlipFlops könnte man sich ähnlich wie bei den in der MMU gebauten Latches von Hand als RS-Latches realisieren. Das Problem mit der Flanken-Triggerung würde sich auch lösen lassen, in dem das Set/Reset-Signal mit sich selbst invertiert AND-verknüpft wird. Als Ergebnis hat man eine Spitze der Länge einer ispLSI-Durchlaufzeit (die nämlich die Invertierung benötigt), die dann zum Setzen bzw. Rücksetzen ausreichen muß. Allerdings ist es mir bei dieser Variante etwas flau in der Magengegend... Obwohl genau dieser Effekt bei flankengetriggerten D-FlipFlops verwendet wird. Aber dort ist auch alles besser aufeinander abgestimmt.
Es mußte also eine bessere Lösung her. Die D-FlipFlops des ispLSI1016 besitzen neben dem Takt- und Dateneingang auch einen programmierbaren Reset-Eingang (siehe auch ispLSI Docu im Anhang). Es existiert also ein flankengetriggerter Eingang (Takt in Verbindung mit D-Eingang) und ein statischer Eingang (Reset). Der dynamische Eingang kann nun für die ATARI-Seite und der statische für die HyperSpeed-Seite verwendet werden. Damit ist das Problem aber noch nicht vollständig aus der Welt geschafft. Denn wenn HyperSpeed mit einem sehr niedrigen Systemtakt betrieben, dann treten die ungewünschten Effekte wieder auf (da dann der XL/XE schneller ist). Deshalb sollte HyperSpeed mit

einem Systemtakt (*DClk*) von mindestens ca. 3.5MHz betrieben werden. Die Herleitung der 3.5MHz (es ist eigentlich noch ein bischen weniger) kann wieder als Übungsaufgabe angesehen werden.
Es gibt zwar auch eine Möglichkeit, die Semaphore 100Einen kleinen Schönheitsfehler gibt es nun noch. Da die HyperSpeed-Seite nur den Reset-Eingang des D-FlipFlops verwenden kann, kann es die full-Semaphore also maximal löschen. Das entspricht aber nicht dem Gewünschtem. Aber wer sagt denn, daß unbedingt für *full*=1 der Buffer als voll gekennzeichnet werden soll? Die Semaphore wird also invertiert verwendet und wird *neg_full* genannt. Überall, wo *full* auftaucht, wird dies nur durch das invertierte *neg_full* substituiert. Dabei ändert sich an der Funktion und Definition von *full* nichts.
Boolesche Gleichungen für die Semaphore:

```
Phase3_b = !PDQ2 & !PDQ1 & !PDQ0;
Phase0_b = PDQ2 & !PDQ1 & !PDQ0;

neg_full.re=F_full_set;
neg_full.clk = EACLK;
reset_point = !A_RW & SDB_Phase1 & Phase3_b
             # A_RW & SDB_Phase1 & Phase0_b;
neg_full.d  =  reset_point
            # !reset_point & neg_full;
```

Da *F_full_set* nur aus einem Produktterm besteht, kann das Reset (bzw. Set) gleich im selben GLB substituiert werden. Mit der steigenden Flanke von *EACLK* wird der Wert von *neg_full* bzgl. der ATARI-Seite modifiziert. *reset_point* gibt dabei mit High an, daß *full* gelöscht bzw. *neg_full* gesetzt werden soll. Ist *reset_point* Low, wird in *neg_full* mit jedem Takt von *EACLK* der alte Wert gespeichert — es ändert sich also nichts. Wann *full* rückgesetzt werden soll, hängt davon ab, ob auf den ATARI geschrieben oder gelesen wird. Im Read-Fall (*A_RW*=1) geschieht dies mit dem Eintritt in *Phase3* bzw. mit dem Ende von *Phase0_b*. Genau zu diesem Zeitpunkt wird bekanntlich auch das gelesene Datum in den Zwischenspeicher übernommen. Wird geschrieben, dann wird der Buffer mit dem Ende von *Phase3* freigegeben.

2.3.4 Effizienzbetrachtungen / Durchsatz

Der maximale Datendurchsatz liegt bei 1.77MB/s (in den Staaten bei 1.79MB/s). Warum dürfte klar sein. Diese obere Schranke wird aber über einen längeren Zeitraum hinweg niemals erreicht, da der ANTIC des XL/XEs immer mit seinen DMA-Zyklen dazwischenfunkt. Um den maximalen Durchsatz zu erreichen ist es notwendig, daß der Zwischenbuffer mehr oder weniger kontinuierlich voll ist. Das bedeudet, daß der Buffer möglichst schnell wieder gefüllt werden muß, nachdem er nach Beendung des letzten ATARI-Zyklus als leer gekennzeichnet wurde.

Zwischenzeiten

Im folgenden gehe ich der Einfachheit halber davon aus, daß die Zeit zum Setzen und Rücksetzen der *full*-Semaphore 0ns ist. In der Praxis kostet aber das Setzen von *full* (bzw. Rücksetzen von *neg_full*) beim verwendeten 90MHz ispLSI1016 im schechtesten Fall etwa 14ns und das Rücksetzen etwa 5ns. Dazu kommen jeweils noch einmal 6.5ns hinzu, bis der neue Wert von *full* an den D-Eingängen der Register liegt.

Zwischen dem Zeitpunkt, an dem der Buffer als leer erklärt wird und dem, zu dem der Buffer spätestens wieder gefüllt sein muß wenn der folgende ATARI-Zyklus genutzt werden soll, liegt ein relativ großer Zeitraum. Dieser Zeitraum unterscheidet sich je nachdem, ob im letzten ATARI-Zyklus geschrieben oder gelesen wurde:

Read: Mit dem Eintritt in *Phase3* (sieh dazu auch noch einmal Abbildung 2.10 auf Seite 33) wird der Buffer auf 'leer' gesetzt. Also bleiben bis zum Beginn von *Phase1* bzw. einen halben *EACLK*-Zyklus später 250ns Zeit.

Write: Hier wird erst mit dem Ende von *Phase3* der Buffer gelöscht. Damit bleiben also nur 180ns übrig (ein *EACLK*-Zyklus weniger).

Dieser Zeitraum kann nun von der HyperSpeed-Seite (inkl. CPU) verwendet werden um den Buffer schnellstmöglich wieder zu füllen. Da aber der HyperSpeed Systemtakt asynchron zu dem vom XL/XE läuft, geht von dieser Zeit im schlechtesten Fall noch ein *DCLK*-Zyklus (35ns bei 14MHz) verloren. D.h. die HyperSpeed-Seite erkennt im schlechtesten Fall erst 35ns später, daß *full* Low ist (weil eben dort *full* unmittelbar nach dem vorherigen *DClk*-Takt auf Low ging und somit die Änderung nicht mehr registriert werden konnte). Weiterhin geht mit Sicherheit beim Lesen noch ein *DClk*-Zyklus verloren. Nämlich der, in dem sich der HyperSpeed-seitige Automat in S4 befindet.
Nun gibt es viele Fälle bzw. Kombinationen von aufeinanderfolgenden Reads und Writes auf den ATARI die auftreten können. Davon möchte ich nur zwei etwas näher betrachten.

Read-only: Dieser Fall tritt z.Bsp. versärkt auf, wenn Code auf dem ATARI abgearbeitet wird. Insgesamt gab es einen Zwischenraum von 250ns. Davon gingen bei 14MHz im schlechtesten Fall 2 *Dclk*-Zyklen, sprich 70ns bei 28MHz Systemtakt ab. Bleiben also noch satte $250ns - 70ns = 180ns$ übrig (im besten Fall 215ns). Wieviel Takte sind das für die CPU, bis *full* wieder gesetzt sein sollte? Bei 14MHz benötigt die 65C816 70ns pro Takt bzw. Maschinenzyklus (ohne Warteschritte). Da wir nur vollständige Takte nehmen können, gehen die 70ns offenbar 2 mal in die 180ns. Also kann hier schon mal mindestens die Pace des ATARI-Bus locker mitgehalten werden. Dazu hätte ja auch nur ein Taktzyklus gereicht. Die Tatsache, daß hier praktisch ein Takt überschüssig ist zieht zwei angenehme Folgen nach sich. Sollte die 65C816 so ein Programm ausführen, das wirklich nur mit jedem Maschinenzyklus ließt, dann nützt der überschüssige Takt nichts. Der wird dann nur im Zustand S5 (Wartezustand beim Lesen) wieder totgeschlagen. In der Realität folgt aber vielen Read-Maschinenzyklen eine interne Operation der CPU. Diese interne Operation wird nun in dem freien Maschinenzyklus abgearbeitet. Im Klartext bedeutet das, daß bei Prorammen, die vollständig auf dem ATARI-Speicher laufen die internen Operationen praktisch nicht mehr ins Gewicht fallen. Grob geschätzt dürfte der Anteil von internen Operationen im Verhältnis zu sämtlichen Maschinenzyklen etwa bei 10-15% liegen (evtl. sogar noch höher). Das bedeutet also, daß diese Programme allein durch diesen Internal Operation SpeedUp um die 10-15% schneller sind. Diese Sache läßt sich aber noch weiter treiben. Wird an den richtigen Stellen anstelle ATARI-Speicher neuer RAM eingeblendet, dann lassen sich auch diverse Adreß- oder Datenreferenzierungen zeitlich gesehen 'wegoptimieren'. Bei einem LDA #$XXXX z.Bsp. kostet dann das eigentliche Laden des Accus praktisch keinen Takt mehr. Allerdings dürfte bei 14MHz CPU-Takt maximal ein halber Warteschritt auf den neuen RAM eingelegt werden, wenn man diesen Effekt jedesmal mit hoher Wahrscheinlichkeit erreichen will. Bei einem halben Warteschritt bleiben von den 180ns nur noch 5ns übrig. Das ist eigentlich schon zu wenig. Denn das Handlen von *full* kostet wie

beschrieben auch seine Zeit. Aber der schlechteste Fall von 180ns muß ja nicht jedesmal eintreten...

Write-only: Wird von der 65C816 permanent auf den ATARI geschrieben (z.Bsp. durch einen Block Move Befehl aus dem neuen RAM in den ATARI), dann macht sich der Zwischenbuffer extrem bemerkbar. Dann steht der CPU sogar noch mehr Zeit zur Verfügung als die 180ns, die nach Beendung eines alten Writes übrig sind. Denn dann kann sie parallel zu einem laufenden ATARI-Zyklus weiterarbeiten. Im schlechtesten Fall muß man davon ausgehen, daß der Buffer unmittelbar bevor *SDB_Phase1* auf 1 geht gefüllt wird. Da die 65C816 hier nicht angehalten werden muß (natürlich nur falls der Buffer leer ist), hat sie also noch zusätzlich die Zeit von Beginn *SDB_Phase1* High bis *full* wieder rückgesetzt wird. Insgesamt ist das ein kompletter ATARI-Zyklus. Diesmal müssen nicht einmal wie beim Lesen zwei Systemtakte für den schlechtesten Fall abgezogen werden. Zum einen muß hier auf nichts gewartet werden und zum anderen geht der HyperSpeed-seitiger Automat beim Schreiben nicht durch einen ähnlichen Zustand wie S4. Die 65C816 hat also (wenn man wieder die Overheads vernachlässigt) einen Spielraum von 560ns. Unter der Bedingung, daß keine Warteschritte eingelegt werden, entspricht das genau 8 65C816 Maschinenzyklen. Die Block Move Befehle der 65C816 benötigen pro bewegtem Byte 7 Maschinenzyklen. Damit ist es also möglich, mittels Software die maximaler Datentransferrate beim Verschieben von Speicherblöcken von HyperSpeed zum ATARI zu erreichen. Umgekehrt ist dies nicht so schnell möglich. Da die 7 benötigten Maschinenzyklen nahezu optimal in die 8 freien Maschinenzyklen passen, ist sogar ein Block Move zum ATARI nur geringfügig langsamer als Block Moves auf dem HyperSpeed-Speicher (natürlich abgesehen von den ATARI-Zyklen die der ANTIC für sich reserviert).

Fazit

Seltene (bzw. wohl verteilte) Schreibzugriffe auf den ATARI machen sich bezüglich Performance-Verlusten überhaupt nicht bemerkbar. Deshalb sollte man sich auch Gedanken machen wie man die Schreibzugriffe geschickt verteilen kann, so daß die CPU möglichst nicht angehalten werden muß. Es sollten also zwischen zwei Schreibzugriffen mindestens zwei bis drei Befehle (je nach deren Länge) liegen, die nicht auf den ATARI zugreifen. Bei Lesebefehlen sieht es im Allgemeinen sowieso schlechter aus. Hier schafft es nicht einmal der Block Move Befehl die ATARI-Zyklen optimal für Nutzerdatentransfers auszunutzen. Von dieser Seite her sollte zwischen zwei Befehlen die jeweils Bytes vom ATARI lesen (z.Bsp. LDA ... LDX ...) ein Befehl stehen, der nicht den ATARI referenziert. Extrem gebremst wird HyperSpeed auch, wenn ein Datum indirekt über einen Pointer referenziert wird und der Pointer liegt auch noch im ATARI-Speicher. Denn dann sind relativ viele Lesezugriffe auf den ATARI notwendig.
Aber noch einmal zur Erinnerung: HyperSpeed wird auch bei Verletzung sämtlicher Regeln immer noch schneller als der alte XL/XE sein.

2.3.5 Mögliche Verbesserungen

Der DataBridge Datendurchsatz zum ATARI hin würde sich extrem beschleunigen lassen, wenn man den Write-Buffer auf etwa 2 bis 4 Stufen ausbauen würde. Momentan besteht zum Beispiel das Problem, daß bei einem 16Bit-Write in den ATARI das erste (niederwertige) Byte schnell in den Buffer übernommen wird. Aber für das unmittelbar folgende (höherwertige) Byte ist erst mal kein Platz im Buffer und somit muß die 65C816 solange angehalten werden, bis wieder Platz im Buffer wird. Ein extrem grösserer Write-Buffer (z.Bsp. 20-30 Bytes) trägt ist schon nicht mehr so sinnvoll. Der würde wahrscheinlich nie voll ausgenutzt — es sei denn durch einen DMA-Controller, der extrem schnell (mit

nahezu jedem Takt) Daten bewegen kann.
Die einzige Möglichkeit Lesezugriffe zu beschleunigen würde darin bestehen, einen Cache zwischen HyperSpeed und ATARI zu setzen. Damit könnten auch Schreibzugriffe mit beschleunigt werden. Aber die große Frage ist, ob dieser Aufwand gerechtfertigt ist. Unabhängig davon müßte man auch die Frage stellen, ob allgemein der Aufwand mit HyperSpeed gerechtfertigt ist...
Abgesehen von diesen prinzipiellen Erweiterungen bei denen auch die DataBridge hardwaremäßig vollkommen umgebaut werden müßte, gibt es meiner Meinung nach auch noch Möglichkeiten die jetzige DataBridge zu verbessern. Und das nur durch einfaches umprogrammieren des ispLSI1016 in dem sich die gesamte Steuerung befindet. Wie Eingangs erwähnt ist bei der HyperSpeed-Seite der Zustand S4 eigentlich überflüssig. Aber wie beschrieben ist nicht klar, ob sich dieser Zustand problemlos entfernen läßt. Beim Rücksetzen und ganz besonders beim Setzen der *full*-Semaphore geht auch sehr viel Zeit verloren. Diese Zeit müßte sich theoretisch auch auf 0 herunterdrücken lassen, indem der jeweilige Automat direkt in den anderen Automaten eingreift — ohne dabei über die seperate *full*-Semaphore zu gehen.
Eine ganz andere Möglichkeit wäre der Einsatz eines höheren Systemtaktes. Derzeit liegt diese bekanntlich bei etwa 28MHz. Da dieser Systemtakt nur für das ispLSI1016 eine Rolle spielt wo die gesamte kritische Taktverwaltung liegt, dürfte eine weitere Erhöhung des Systemtaktes keine so großen Probleme bereiten. Bei HyperSpeed V2.1 kommt standardmäßig eine 90MHz Version des ispLSI1016 zum Einsatz (Momentan sind von Lattice auch 125MHz Versionen erhältlich, aber der Preis...). Diese 90MHz lassen sich hier nicht voll ausschöpfen, da diese nur bei der internen Kommunikation und beim 4 Productterm Bypass (siehe auch ispLSI Docu) erreicht werden. Aber so 80MHz dürften drin sein. Eine weitere Verdopplung des Systemtaktes auf etwa 56MHz liegt also auf der Hand. Wird dann der HyperSpeed-seitige DataBridge Automat mit diesen 56MHz getaktet, dann halbiert sich die schlechtste Reaktionszeit auf die Änderung von *full* von derzeit maximal 35ns auf 18ns. Damit wird die DataBridge zwar nur minimal schneller, aber manchmal kann jede Nanosekunde zählen. Die CPU wird aber nach wie vor noch mit 14MHz getaktet. Abgesehen von der DataBridge könnte man bei dieser Gelegenheit auch gleich viertel-Warteschritte einführen...

2.4 ATARI Control

Neben der DataBridge gibt es noch weitere Möglichkeiten in den Steuerpfad des XL/XEs einzugreifen bzw. den Einfluß des XL/XEs auf die Funktion von HyperSpeed zu steuern.

2.4.1 Reset

HyperSpeed bekommt das *Reset*-Signal vom ATARI. Diese Tatsache ist der einzige (eigenlich lächerliche) Grund dafür, daß HyperSpeed nicht so richtig als stand-alone Rechner existieren kann. Bei späteren Versionen wird dieser Mißstand aber wahrscheinlich durch einen eigenen PowerOn-Reset Controller beseitigt sein. Das vom XL/XE ankommende *Reset*-Signal wird auf HyperSpeed erst einmal über zwei Inverter verstärkt.

2.4.2 IRQ, NMI und RDY

Diese vom XL/XE kommenden Steuersignale können alle einzeln mittels Softwareschalter entweder zu HyperSpeed durchgelassen oder abgeblockt werden. Tabelle 2.8 zeigt die entsprechenden VIA-Bits, die für die einzelnen Signale zuständig sind. Ist ein Steuerbit auf 1, dann ist die zugehörige Aktion

Tabelle 2.8: VIA Enable-Bits für IRQ, NMI und RDY

Signal	VIA Bit
RDY	Port A / Bit 2
IRQ	Port A / Bit 3
NMI	Port A / Bit 4

disabled. Alle drei Control-Signale haben jeweils einen PullUp Widerstand, so daß unmittelbar nach einem Reset weder *IRQs* noch *NMIs* vom ATARI zu HyperSpeed kommen und daß ein Anhalten der HyperSpeed CPU durch *RDY* vom ATARI aus nicht möglich ist. In der Praxis geschieht das Enablen/Disablen der Signale durch einfaches verodern des Enable-Signals mit den vom XL/XE kommenden Signalen (und anschließender Konvertierung in ein OpenCollector Signal).
Ein Abkoppeln der Interrupts ist zum Beispiel dann sinnvoll, wenn die alte CPU und HyperSpeed parallel laufen. Im Allgemeinen macht es wenig Sinn, daß bei der ATARI und HyperSpeed CPU gleichzeitig Interrupts ausgelöst werden. In manchen Fällen kann dies aber wiederum sehr hilfreich sein.
Interrupts, die von HyperSpeed-Einheiten oder von HyperSpeed-Erweiterungen ausgelöst werden, werden **nicht** auf den ATARI weitergeleitet!

2.4.3 Steuerung der alten CPU

Die ATARI CPU kann über zwei Möglichkeiten von HyperSpeed aus über einen längeren Zeitraum angehalten werden: Einmal über die *Halt*-Leitung und zum anderen über die *RDY*-Leitung der CPU. Damit sind aber wirklich nur die Pins an der CPU gemeint. Denn die beiden Leitungen vom XL/XE-Board zur alten CPU müssen durchtrennt werden. Dafür müssen diese beiden Signale vom XL/XE-Motherboard zu HyperSpeed geführt werden, dort werden die entsprechenden CPU-Steuersignale eingemischt und es geht wieder zurück zur alten CPU. Dabei ändert sich aber prinzipiell nichts am Verhalten der alten CPU (wenn sie nicht gerade angehalten wurde...). Der Nachteil der ganzen Sache ist der, daß etwas härter in den ATARI eingegriffen werden muß, und daß die entsprechenden Signale über ein extra Kabel ausgetauscht werden müssen. Sollte HyperSpeed aus dem System wieder entfernt werden, dann muß nur mit zwei Drahtbrücken CPU-RDY und Board-RDY sowie CPU-Halt und Board-Halt auf dem Steckverbinder wieder verbunden werden (siehe auch HyperSpeed PCB im Anhang). Sollte die alte CPU aus dem ATARI entfernt werden, dann entfällt das Durchtrennen der Leitungen.
Zurück zum Thema. Wird ein maximaler Datendurchsatz der DataBridge benötigt, dann muß die alte CPU über die *Halt*-Leitung angehalten werden. Dazu gibt es das *Hlt_A*-Signal, welches durch Bit 1 / Port A des VIAs repräsentiert wird. *Hlt_A* ist Low-aktiv und besitzt einen PullUp Widerstand. Also wird die alte CPU nach einem Reset nicht über *Halt* angehalten. Das hat auch einen ganz bestimmten Grund. Denn ein Anhalten der CPU über *Halt* hat zur Folge, daß intern der CPU-Takt auf Low gehalten wird. Die Register alter (teilweise auch neuerer) CPUs sind meist dynamisch aufgebaut, d.h. sie müssen wie DRAMs immer aufgefrischt werden. Bekommt die CPU nun sehr lange keinen Takt (wie lange ist wieder mal nur durch experimentieren herauszubekommen), dann können die Registerinhalte verlorengehen was einen Absturz der CPU zur Folge hat. Aus diesem Grund ist es auch nicht ratsam, die alte CPU von HyperSpeed aus über einen längeren Zeitraum über *Hlt_A* zu stoppen, wenn sie in nächster Zukunft ohne ein Reset wieder aktiviert werden soll. Sollte die CPU in diesem Fall

irgendwann wieder freigegeben, dann wird es mit hoher Wahrscheinlichkeit Probleme geben. Nebenbei bemerkt ist die bei HyperSpeed verwendete CPU von WDC ist voll statisch aufgebaut und besitzt dieses Manko glücklicherweise nicht.
Ein Anhalten der alten CPU über deren *RDY*-Leitung geschieht über die Low-aktive *STP_A*-Leitung (Bit 0 / Port A des VIAs). Diese Leitung hat einen PullDown Widerstand und ist somit unmittelbar nach einem Reset aktiv. Bei meinen Tests ist es dabei allerdings zu Problemen gekommen. Denn genau genommen wird *STP_A* nicht nach einem Reset aktiv, sondern nachdem *Reset* aktiv wurde. Das hat aber offenbar der alten CPU nicht sonderlich gefallen. Denn nachdem *RDY* wieder freigegeben wurde hat sich die alte CPU sehr oft aufgehangen. Die Vermutung lag nahe, daß es Probleme gibt weil während *Reset* noch aktiv ist *RDY* schon aktiv wird. Aus diesem Grund wird ein D-FlipFlop zwischengeschaltet, welches mit der *Phi0S* immer updated wird und durch den asynchronen (Low-aktiven) Set-Eingang gesetzt wird, falls *Reset* aktiv ist. Also wird während des Reset-Vorgangs vermieden, daß *RDY* aktiv ist. In der Praxis scheint sich dies auch als Lösung des Problems zu bestätigen.
Soll die alte CPU über einen längeren Zeitraum angehalten werden mit der Möglichkeit sie irgendwann weiter laufen zu lassen, dann sollte das über *STP_A* abgewickelt werden. Denn mit *RDY* wird nur der interne Control-Automat der CPU in seinem aktuellen Zustand gehalten und es gibt keine Verluste von Registerinhalten.

2.4.4 ATARI Clocks

Als CPU-Takt wird auf dem ATARI die *Phi0* bzw. global gesehen die *Phi0S* erzeugt. Die ATARI-CPU erzeugt ihrerseits wieder eine *Phi1S* und eine *Phi2S*. Die *Phi1S* ist die invertierte *Phi0S* und ist wie die *Phi2S* etwa 20ns bezüglich der fallenden Flanke der *Phi0S* nach hinten verschoben. Das dürfte aber auch je nach Version mehr oder weniger stark variieren. Der Zeitversatz nach der steigenden Flanke der *Phi0* ist noch etwas größer. Wird die alte CPU aus dem ATARI entfernt, dann müssen also die beiden Takte *Phi1S* und *Phi2S* dem restlichen XL/XE-System zur Verfügung gestellt werden. Wie dies im einzelnen geschieht ist dem Schaltplan (ATARI Interface) zu entnehmen. Viel ist diesem nicht mehr hinzuzufügen. Die Verzögerungen erfolgen mit den HCT-Invertern (typ. Verzögerung 10ns). Mit dem Jumper J10 wird die auf HyperSpeed erzeugte *Phi2S* auf den ATARI gebracht und mit J19 entsprechend die *Phi1S*. Mittels Jumper J2 kann noch optional die Verzögerung des ACT-AND Buffers mit in die Kette zur Bildung der Takte aufgenommen werden (Jumperstellung 2-3). Sollten die Jumper J10 oder J19 gesteckt sein, trotz daß sich die alte CPU im XL/XE befindet, dann ist das nicht allzu schlimm. Dann werden zwar die Ausgänge der HCT-Inverter mit den Ausgangstreibern der CPU zusammengeschalten, aber erstens sind die HCT-Gatter nicht sehr stark und zweitens laufen ja die jeweils zusammengeschaltenen Ausgänge mehr oder weniger genau in Phase. Schaltungstechnisch ist dieser Zustand nicht illegal, sondern dies hat eher einen Verstärkungseffekt zur Folge. In manchen Fällen wird evtl. dieser Effekt sogar benötigt.

Anhang A

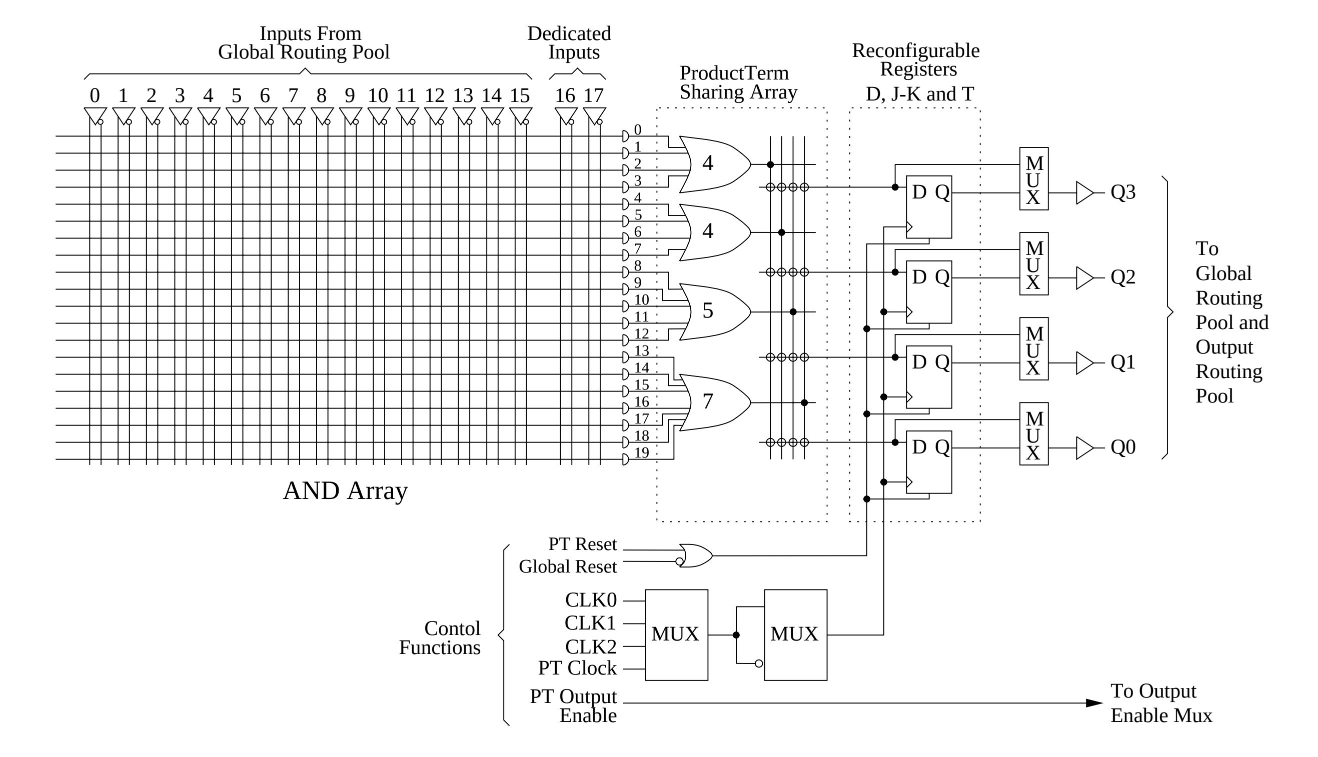
ispLSI Serie von Lattice

Es folgt eine kurze Beschreibung der ispLSI Serie von Lattice. Diese Beschreibung ist eher unvollständig. Es soll nur in etwa das Prinzip herüberkommen. Für nähere Beschreibungen wird auf einschlägige Literatur verwiesen.

A.1 Was ist die ispLSI Serie

'ispLSI' steht für in system programable Large Scale Integration. Das 'in System' ist für HyperSpeed von untergeordneter Bedeutung. Dadurch ist es aber möglich den ChipSatz bei evtl. Änderungen direkt on Board, d.h. ohne ein spezielles Programmiergerät neu zu programmieren.
Die ispLSI Architektur kennzeichnet sich durch eine Menge von gleichartigen sog. Megablocks und durch einen *Global Routing Pool — GRP*. Das ispLSI1016 hat zwei Megablöcke und das ispLSI2032 einen. Jeder Megablock besteht wieder aus 8 Logikblöcken (sog. *GLBs — Generic Logic Blocks*), 16 IO Cells (bei der 2000er Serie 32 IO Cells) und zwei Dedicated Inputs. Weiterhin gibt es noch ein paar globale Takte und diverse andere Signale. Ein GLB der 1000er und 2000er Serie besitzt 18 Eingänge, die invertiert und nich invertiert auf eine AND-Matrix geführt werden. Ein ProductTerm kann also aus maximal 18 Signalen bestehen. 16 der 18 Signale kommen vom *GRP*, zwei von speziellen Eingängen, sog. Dedicated Inputs.
Ein GLB besitzt 4 Ausgänge. Diesen können wahlweise D-FlipFlops vorgeschaltet werden. Die 4 Ausgänge ergeben sich aus der 'VerORung' von ProductTerms. Dabei sind maximal 20 ProductTerms möglich. Diese 20 teilen sich auf in eine Kombination von 4 + 4 + 5 + 7. Dabei besteht aber die Möglichkeit mehrere von den vier primär-ORs zusammenzufassen.
Weiterhin können noch nach den OR-Gattern vor den Ausgang (bzw. vor den Eingang des D-FlipFlops) geschaltet werden. Diese XORs können wahlweise nur mit einem ProductTerm oder mit einem ProductTerm und einem Ausgang eines primären ORs angesteuert werden.
Wird für ein Signal eine besonders schnelle Behandlung benötigt, kann man ein sog. 4 ProductTerm Bypass verwenden.
Der Takt der D-FlipFlops kann entweder von einem Systemtakt (zum Beispiel von einem speziellen externen Pin) oder aus einem ProductTerm bezogen werden. Leider gibt es in einem GLB für alle FlipFlops nur einen Takt. Das Reset der FlipFlops kommt von einem externen Pin. Wahlweise kann dafür auch noch ein (High-aktives) ProductTerm Reset verwendet werden, welches mit in das globale Reset gemischt wird.
In einem GLB kann desweiteren ein ProductTerm als Output Enable für Bidirektionale IO Cells des zugehörigen MegaBlocks verwendet werden.

Abbildung A.1: GLB der 1000er Serie — Prinzipschaltung



Die Ausgänge werden auf den GRP gebracht und können auch einer IO Zelle zugeführt werden. Auf dem Weg zur IO Zelle kann dabei noch der *Output Routing Pool — ORP* genutzt werden. Damit können Änderungen am Pinout relativ einfach angepaßt werden. Da der ORP auch seine Zeit kostet, gibt es einen ORP Bypass. Dabei gibt es eine feste Zuordnung der GLB Outputs zu den IO Cells. Bild A.1 zeigt die Prinzipschaltung eines GLBs der 1000er Serie in der Normalkonfiguration (ohne XOR-Gatter, ohne 4 ProductTerm Bypass). Ein GLB der 2000er Serie hat lediglich ein paar Einschränkungen bei den Control Functions.
Die IO Cells können wahlweise als Input, Output oder Bidirektional betrieben werden. Bei der 1000er Serie können sogar an die Eingänge noch D-Latches oder D-FlipFlops geschaltet werden (beim bidirektionalen Pin Eingangsseitig nur D-FlipFlops).
Alle als Input deklarierten IO Cells werden auf den GRP gebracht. Über diesen werden also sämtliche Eingänge (außer Dedicated Inputs, Clocks, Reset) und Feedbacks zu den GLBs gebracht. Das verwendete ispLSI1016 verfügt über zwei MegaBlocks (16 GLBs, 32 IO Cells) und das ispLSI2032 über einen MegaBlock (8 GLBs, aber 32 IO Cells).
Das dürfte für das grobe Verständnis ausreichen...

A.2 pDS Syntax

Listings bzw. Auszüge aus Listings für die ispLSI Serie von Lattice basieren auf einer speziellen Sprache, die das pDS (Entwicklungssoftware für die ispLSI Serie) verdaut. Damit diese Listings für Außenstehende nicht vollkommen unlesbar sind, folgt hier eine kleine Beschreibung — zumindest von den Sachen, die hier verwendet wurden.

Tabelle A.1: Boolesche Operatoren

Operator	Vorrang	Operation
()	0	Klammerung
!	1	Inversion
&	3	AND
#	4	OR
=		Zuweisung

Tabelle A.2: GLB Dot Extensions

.d	D-Eingang eines FlipFlops
.q	Q-Ausgang eines FlipFlops - Feedback
.pin	Feedback eines kombinatorischen Ausgangs
.re	High-aktives ProductTerm Reset (wirkt auf alle FlipFlops im entsprechenden GLB)
.clk	Ein Systemtakt wird für alle FlipFlops im GLB verwendet
.ptclk	Ein entsprechender ProductTerm Clock wird als Takt für alle FlipFlops im GLB verwendet
.oe	Das entsprechende Signal ist ein High-aktives Output Enable (ein ProductTerm) welches den IO Cells zugeführt wird

A.2.1 Boolesche Operatoren

Tabelle A.1 zeigt einen Subset der möglichen booleschen Operatoren. Einige Signale werden mit einem Punkt und einer Extension. In Tabelle A.2 sind die Bedeutungen dieser Extensions zu finden.

A.2.2 GLBs

GLBs werden im Listing mit *SYM GLB ...* eingeleitet und mit *END;* beendet. In jedem GLB werden zuerst die Ausgangssignale durch eine *SigType* Anweisung deklariert. Dieser Anweisung folgt der Signalname und und ein oder mehrere Attribute. Tabelle A.3 zeigt ein paar dieser Attribute. Als nächstes folgt die Beschreibung der Logik. Diese wird mit *EQUATIONS* eingeleitet und mit *END;* beendet.

A.2.3 IO Cells

Im Listing werden IO Zellen mit *SYM IOC ...* eingeleitet und mit *END;* beendet. In den IO Zellen werden externe Signale durch eine *XPin* Anweisung deklariert. Der *XPin* Anweisung folgt als erstes der Typ der IO-Zellen, dann der Name des externen Signals und zum Schluß entsprechende Attribute. Tabelle A.4 zeigt die möglichen Zelltypen und Tabelle A.5 mögliche Attribute der externen Signale.

Tabelle A.3: Signalattribute im GLB

async	Damit wird dem Router mitgeteilt, daß es sich um ein asynchrones Signal handelt. Normalerweise legt der Router bei Routing Problemen manchmal ein Signal mehrfach an (Aliasing). Bei asynchronen Signalen kann dies mitunter fatale Folgen haben. Das async-Attribut verhindert das Aliasing.
critical	Weist den Router an, für das entsprechende Signal den 4 Product-Term Bypass zu verwenden.
out	Deklariert das Signal als Ausgang des GLBs. Dieses Attribut besitzen prinzipiell alle Signale, die innerhalb eines GLBs nicht deklariert wurden. Solche Signale werden aber bei HyperSpeed nicht verwendet. Alle Signale, die nicht mittels SigType deklariert wurden, werden innerhalb des GLBs direkt substituiert. Sie existieren nur aus Übersichsgründen.
reg	Dieses Attribut deklariert das Signal als Ausgang eines D-FlipFlops.

Tabelle A.4: Mögliche Typen von IO Zellen

IO	Das Signal befindet sich in einer normalen IO Zelle
I	Das Pin bezieht sich auf einen *Dedicated Input* — also nur als Eingang zu verwenden
CLK	Das Pin bezieht sich auf einen *Dedicated Clock Input*
GOE	Pin bezieht sich auf ein Globales Enable; nur bei der 2000er Serie

Tabelle A.5: Attribute für externe Signale

lock	Das entsprechende Signal ist fest an das Pin der verwendeten Package (bei HyperSpeed PLCC) gelockt. Ein locken von Signalen an bestimmte Pins schränkt den Freiheitsgrad des Routers ein.
critical	Für das Signal wird der Output Routing Pool umgangen (ORP Bypass). Das schränkt auch die Routbarkeit des Desings ein, ist aber für kritische Signale besser.
slowslew	Nur für die 2000er Serie. Dieses Attribut hat zur Folge, daß die Slewrate des Ausgangs geringfügig veringert wird. Der Ausgang schaltet also nicht so schnell um. Dies ist für bestimmte Signale bzgl. Reflexionsverringerung nützlich.
pullup	Damit kann ein PullUp Widerstand am externen Pin aktiviert werden.

Tabelle A.6: Einige Makros

ob11	normaler kombinatorischer Ausgang
ob21	invertierter Ausgang
ib11	normaler Eingang
bi11	bidirektionales Pin

A.2.4 Makros

Das pDS System unterstützt auch die Verwendung von Makros. Tabelle A.6 beschreibt bzw. benennt einige Makros, die in den IO Zellen verwendet wurden. Wie diese Makros verwendet werden dürfte aus dem Kontext heraus erkennbar sein.

Anhang B

Die 65C816 CPU

Hier soll nur das Nötigste und 65C816-spezifische beschrieben werden, um mit der CPU umgehen zu können. Ein Großteil ist ohnehin identisch zur 6502. Für detailliertere Informationen sei auf die Datenblätter von WDC verwiesen. Dies ist mehr oder weniger ein Auszug bzw. eine Übersetzung dieser Datenblätter. Dabei habe ich mir gleich mal das Recht herausgenommen und einige Fehler mit berichtigt. Die CPU wird übrigens vom derzeitig erhältlichen 'MAE-Assembler' von John Harris unterstützt.

B.1 Definitionen

B.1.1 Bank

Die 65C816 besitzt einen linearen Adressraum von 16MB. Demzufolge wird eine 24Bit Adresse benötigt. Die oberen 8Bit dieser Adresse stellen die Bank dar. Es gibt also 256 Bänke zu je 64kB.

B.1.2 CPU Modi

Die CPU besitzt einen Emulation Mode und einen Native Mode. Nach einem Reset befindet sich die CPU im Emulation Mode. In diesem Mode sind schon nahezu uneingeschränkt alle neuen Befehle und Adressierungsarten nutzbar. Eine volle Ausnutzung der CPU ist aber erst im Native Mode möglich (16Bit Register, BlockMove Befehle).

B.1.3 Word Adressierung

Sämtliche Bezüge auf Daten, die größer als ein Byte sind, laufen bei der 65C816 wie bei der 6502 nach dem Low/High Format ab. Das bedeutet also, daß sich in der referenzierten Adresse das Low-Byte befindet und eine Adresse weiter das High-Byte. 16Bit Words oder 24Bit Words müssen dabei nicht notwendigerweise an einer geraden Adresse im Speicher beginnen wie dies bei einigen anderen Prozessoren der Fall ist. Werden 24Bit Daten bzw. Adressen auf dem Stack abgelegt, so wird zuerst das höherwertige und zum Schluß das niederwertige Byte abgelegt.

B.2 Register

B.2.1 Data Bank Register (DBR)

DBR ist ein 8Bit Register, welches bei der bekannten Absoluten Adressierungsart die oberen 8Bit der Adresse des adressierten Bytes enthält (siehe Adressierungsarten Seite 53).

B.2.2 Program Bank Register (PBR)

Die 65C816 besitzt wie die 6502 auch nur einen 16Bit Program Counter. Die oberen 8Bit der Programmadresse bildet deshalb PBR. Das hat Vor- und Nachteile. Zum Beispiel ist es nicht möglich, den Programmcode einfach hintereinanderweg über mehrere 64kB-Bänke hinweg zu spannen. Durch die bankorientierte Adressierung ist z.B. das Code-Folgebyte von $01FFFF nich $020000, sondern $010000. Ein Vorteil der Bankorientierung ist, daß mehrere für einen 16Bit *absoluten* Adressraum geschriebene Programme in verschiedene Bänke (die ja jeweils einen 16Bit Adressraum darstellen) geladen werden können.

B.2.3 Direct Register (D)

Dieses 16Bit Register steht in unmitelbaren Zusammenhang mit der Zero-Page der 6502. Bei der 65C816 gibt es keine Zero-Page in dem Sinn mehr. Alle Adressierungsarten der 6502, die sich auf die Zero-Page beziehen, laufen jetzt über das Direct Register ab. Steht D zum Beispiel auf $0000, hat das den selben Effekt wie die Zero-Page Adressierung der 6502 (also $0000+Offset). Steht D auf $0600, wird $0600+Offset adressiert. Das Direct-Konzept stellt sozusagen eine verschiebbare Zero-Page dar. Eine Direct Adressierung bezieht sich immer auf Bank $00.
Nach einem Reset steht D auf $0000.

B.2.4 Stack Pointer (S)

Der Stackpointer ist auf 16Bit erweitert worden. Der Stack wird genau wie bei der 6502 adressiert und liegt generell in Bank $00. Im 6502-Emulation Mode steht S auf $0100. Das höherwertige Byte von S kann dabei im Emulation Mode nicht verändert werden!

B.2.5 Accu (C=A+B)

Der Accu kann im Native Mode wahlweise mit 16Bit (C) oder 8Bit (A,B) Breite betrieben werden. Es gibt noch einen Befehl (XBA - Exchange B and A Accumulator), der die beiden Accuhälften A und B vertauscht. Somit hat man praktisch auch im Emulation Mode ein zusätzliches 8Bit Register.

B.2.6 Index Register (X und Y)

X und Y können im Native Mode entweder beide 16Bit oder beide 8Bit breit sein. Im Emulation Mode sind beide 8Bit breit. Das höherwertige Byte von X und Y ist im Emulation Mode immer $00.

B.2.7 Processor Status Register (P)

P ist wie gehabt noch 8Bit breit und hat folgenden Aufbau:

7	6	5	4	3	2	1	0	Mode
		1	B				E	Emulation (E=1)
N	V	M	X	D	I	Z	C	Native (E=0)

Hier hat sich also nicht viel geändert. Im Native Mode gibt es die beiden neuen Flags M und X. Diese beiden Bits geben die Breite der Indexregister X und Y und des Accus an.

- M=0 : Accu 16Bit
  M=1 : Accu 8Bit (höherwertiges Byte vom Accu bleibt lediglich 'versteckt', kann aber mit XBA mit dem niederwertigen vertauscht werden)
- X=0 : X und Y 16Bit
  X=1 : X und Y 8Bit (höherwertiges Byte ist jeweils $00)

Das Break Flag ist nicht mehr direkt zugänglich. Tritt aber ein Interrupt auf und P wird im Stack abgelegt, wird anstelle des X-Flags das B-Flag mit abgelegt.

B.3 Addressierungsarten

Die 65C816 kennt 24 verschiedene Adressierungsarten (die 6502 nur 11). Folgende Adressierungsarten generieren 24Bit effektive (Daten-)Adressen:

- Direct Indexed Indirect (d,x)
- Direct Indirect Indexed (d),y
- Direct Indirect (d)
- Direct Indirect Long [d]
- Direct Indirect Long Indexed [d],y
- Absolute a
- Absolute Indexed a,x
- Absolute Indexed a,y
- Absolute Long al
- Absolute Long Indexed al,x
- Stack Relative Indirect Indexed (d,x),y

Die restlichen Adressierungsarten sind Spezialadressierungsarten.

B.3.1 Immediate Addressing

Der Operand ist das zweite (und dritte bei 16Bit) Byte des Befehls.

B.3.2 Absolute a

Das zweite und dritte Byte des Befehls bildet die 16Bit Adresse (wie bei der 6502). Das Daten Bank Register DBR enthält die oberen 8Bit.

Instuction:	OpCode	addrl	addrh
Operand Address:	DBR	addrh	addrl

B.3.3 Absolute Long al

Alle drei Bytes der 24Bit Adresse befinden sich im Befehl (natürlich im Low-High Format).

Instuction:	OpCode	addrl	addrh	baddr
Operand Address:	baddr	addrh	addrl	

B.3.4 Direct d

Aus der Addition des zweiten Bytes des Befehles und des Direct Registers ergibt sich die 16Bit Adresse des Operanden. Die Bank ist dabei immer $00. Bei jeder Referenzierung auf das Direct Register wird übrigens ein zusätzlicher Zyklus benötigt, falls das niederwertige Byte von D nicht $00 enthält.

Instuction:	OpCode	offset	
		Direct Register	
	+		offset
Operand Address:	00	effective address	

B.3.5 Accumulator A

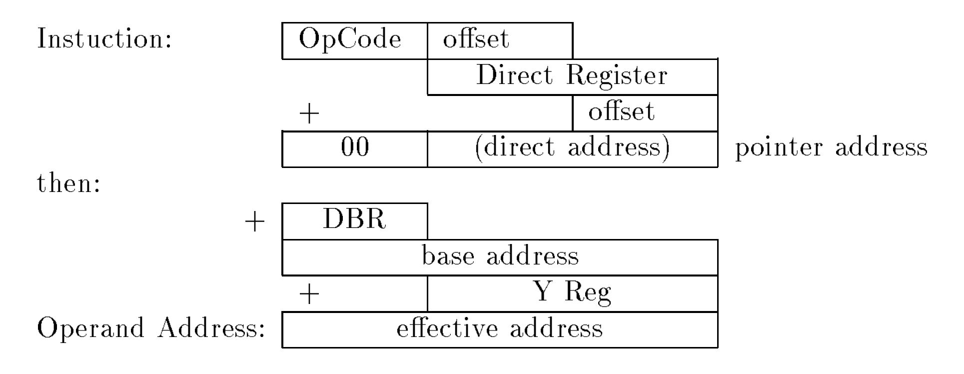
Single Byte Instruction. Operation nur auf dem Accu.

B.3.6 Implied i

Single Byte Instruction. Der Operand ist von der Operation abhängig.

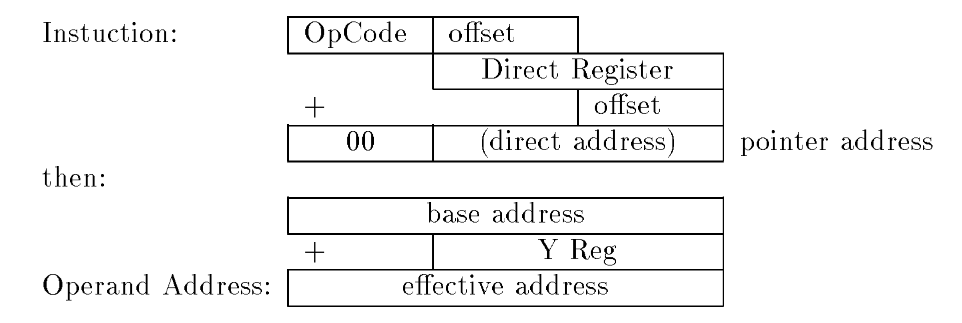
B.3.7 Direct Indirect Indexed (d),y

Über die durch die Addition des Direct Registers und des zweiten Bytes erhaltenen 16Bit Addresse wird in Bank $00 ein 16Bit Pointer referenziert. Aus der Addition dieses Pointers und des Y Registers ergibt sich in Verbindung mit DBR die 24Bit Operandenadresse.



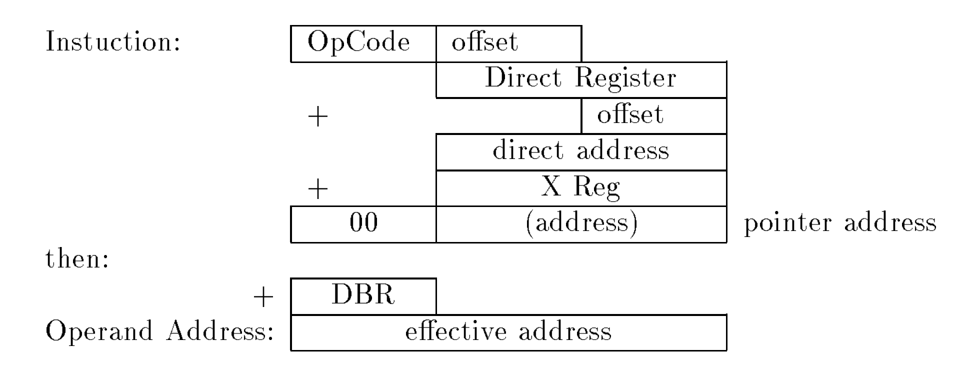
B.3.8 Direct Indirect Long Indexed [d],y

Aus der Summe des zweite Bytes des Befehls und des Direct Registers ergibt sich eine 16Bit Adresse, die auf einen 24Bit Pointer in Bank $00 zeigt. Aus der Summe des 24Bit Pointers und des Y Registers ergibt sich die 24Bit Operandenadresse.



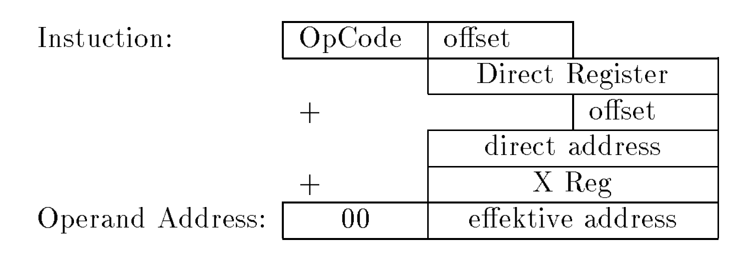
B.3.9 Direct Indexed Indirect (d,x)

Die aus der Summe des zweiten Bytes, des X Registers und des Direct Registers erhaltene 16Bit Adresse zeigt auf einen 16Bit Pointer in Bank $00. Anhand dieses Pointers und DBR ergibt sich die effektive 24Bit Adresse.



B.3.10 Direct Indexed With X d,x

Das zweite Byte des Befehls und das X Register werden zum Direct Register hinzuaddiert. Das Ergebnis ist die Effektive Adresse. Der Operand liegt dabei stets in Bank $00.



B.3.11 Direct Indexed With Y d,y

Wie d,x – nur mit dem Y Register.

Instuction:	OpCode	offset	
		Direct Register	
	+		offset
		direct address	
	+	Y Reg	
Operand Address:	00	effektive address	

B.3.12 Absolute Indexed With X a,x

Das zweite und dritte Byte des Befehls bilden die niederwertigen 16Bit der Basisadresse. Dazu wird das X Register addiert. Die Bank des Operanden bildet DBR.

Instuction:	OpCode	addrl	addrh
	DBR	addrh	addrl
	+	X Reg	
Operand Address:	effektive address		

B.3.13 Absolute Long Indexed With X al,x

Im Befehl steht die gesamte 24Bit Basisadresse. Zu dieser wird X hinzuaddiert.

Instuction:	OpCode	addrl	addrh	baddr
	baddr	addrh	addrl	
	+	X Reg		
Operand Address:	effective address			

B.3.14 Absolute Indexed With Y a,y

Wie a,x – nur mit dem Y Register.

Instuction:	OpCode	addrl	addrh
	DBR	addrh	addrl
	+	Y Reg	
Operand Address:	effektive address		

B.3.15 Program Counter Relative r

Das zweite Byte des (Branch-) Befehls wird im Falle einer Verzweigung zum Program Counter addiert. Der Offset ist dabei vorzeichenbehaftet (also -128 bis +127). Achtung! Es findet wie gesagt kein übertrag in die nächste Bank statt.

B.3.16 Program Counter Relative Long rl

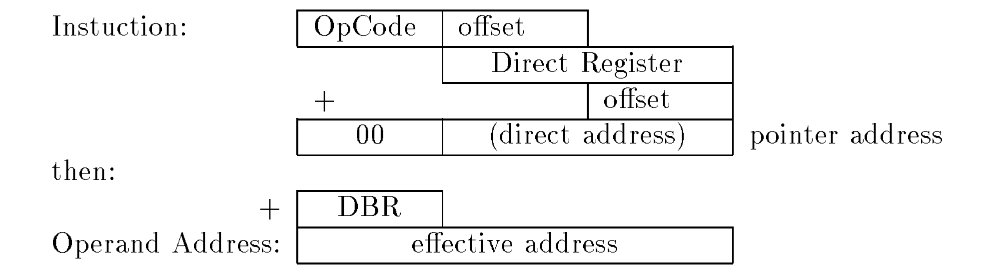
Das zweite und dritte Byte des Befehls (Low/High Format) bildet einen 16Bit Offset (-32768 bis 32767). Diese Adressierung wird nur von BRL (unconditional BRanch Long) verwendet. Damit sind also relative Sprünge über einen 16Bit Bereich möglich. Hier findet auch kein Übertrag in die nächste Bank statt!

B.3.17 Absolute Indirect (a)

Das zweite und dritte Byte des Befehls zeigen auf einen Jump Pointer in Bank $00. Der Program Counter wird mit dem ersten und zweiten Byte dieses Pointers geladen (beim JMP und JML (JuMp Long) Befehl). Bei JML wird noch das dritte Byte des Pointers in das Program Bank Register PBR geladen. JMP und JML sind die einzigen Befehle, die diese Adressierung verwenden.

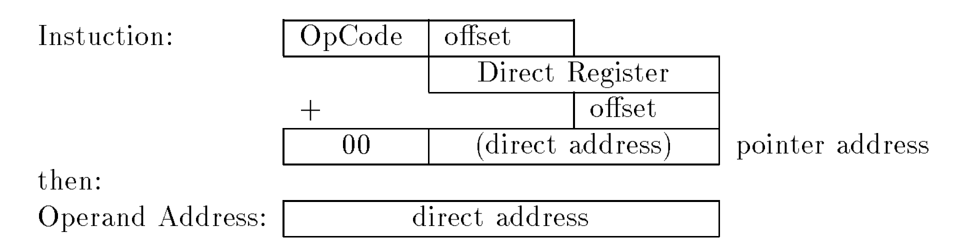
B.3.18 Direct Indirect (d)

Die Addition des zweiten Bytes und des Direct Registers ergibt eine 16Bit Adresse eines 16Bit Pointers in Bank $00. Der 16Bit Pointer in Verbindung mit DBR bildet die effektive Operandenadresse.



B.3.19 Direct Indirect Long [d]

Die Addition des zweiten Bytes und des Direct Registers ergibt eine 16Bit Adresse eines 24Bit Pointers in Bank $00, welcher die Adresse des Operanden enthält.



B.3.20 Absolute Indexed Indirect (a,x)

Das zweite und dritte Byte des Befehls bildet eine 16Bit Basisadresse der aktuellen Program Bank. Zu dieser Basisadresse wird das X Register hinzuaddiert. Das Ergebnis zeigt auf einen 16Bit Wert in Bank PBR, welcher in den Program Counter geladen wird. PBR ändert sich nicht!

B.3.21 Stack s

Die Stackadressierung bezieht sich auf alle Befehle die Daten auf dem Stack ablegen oder vom Stack holen (Push, Pull, JSR, RTS, Interrupts, RTI). Der Stackbereich liegt immer in Bank $00.

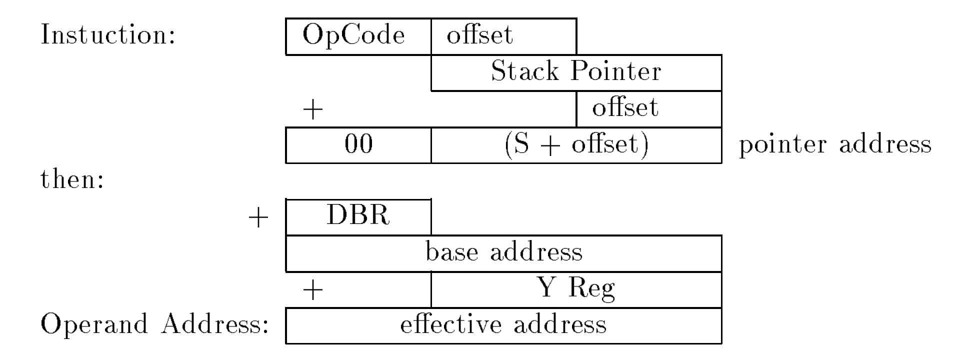
B.3.22 Stack Relative d,s

Die Addition des zweiten Bytes (als Offset im Bereich von 0 bis 255) und des Stackpointers ergibt die 16Bit Adresse des Operanden in Bank $00.

Instuction:	OpCode	offset	
		Stack Pointer	
	+		offset
Operand Address:	00	effective address	

B.3.23 Stack Relative Indirect Indexed (d,s),y

Die Addition des zweiten Bytes (wieder als Offset zwischen 0 und 255) und des Stack Pointers ergibt diesmal die 16Bit Adresse eines 16Bit Pointers in Bank $00. Aus der Addition dieses Pointers und dem Y Register ergibt sich in Verbindung mit DBR die 24Bit Operandenadresse.



B.3.24 Block Source Bank, Destination Bank xyc

Diese Adressierung wird ausschließlich für die Block Move Befehle verwendet. Das zweite Byte enthält die Ziel-Bankadresse und das dritte Byte die Quell-Bankadresse. Das X Register enthält die niederwertigen 16Bit der Quelladresse und das Y Register die niederwertigen 16Bit der Zieladresse. Der C Accu (also 16Bit) enthält die Anzahl der zu bewegenden Bytes minus 1. Achtung! DBR wird mit der Zielbankadresse überschrieben! Pro bewegtem Byte wird der Accu dekrementiert und das X und Y Register inkrementiert (MVN - block MoVe Negative) oder dekrementiert (MVP - block MoVe Positive). Ist der Accuinhalt nach einem bewegtem Byte größer als 0 (1 bis 65535), werden vom Program Counter 3 abgezogen — der OpCode wird praktisch neu gelesen. Das hat einen Vor- und Nachteil. Der Vorteil ist, daß evtl. während eines Move Befehls auftretende Interrupts relativ schnell erfaßt werden können und somit ein Move Befehl kurz unterbrochen und später fortgesetzt wird. Der Nachteil ist, daß dadurch pro bewegtem Byte 7 Taktzyklen notwendig sind. Es ist aber meinerseits bereits ein DMA Controller mit einem Block Move Processor in Vorbereitung. Dieser soll dann für ein Word (momentan noch ein Byte groß) nur noch zwei Takte benötigen...

Instruction:	OpCode	dstbnk	srcbnk
	DBR := dstbnk		
Source Address:	srcbnk	X Reg	
Dest. Address:	DBR	Y Reg	

Increment (MVN) or decrement (MVP) X and Y.
Decrement C (if greater than zero), then PC := PC-3.

B.4 Der Befehlssatz der 65C816

Tabelle B.1 zeigt alle Befehle der 65C816.

B.4.1 Nähere Erläuterungen

Hier sollen nur ein paar neue Befehle diskutiert werden. Alles übrige ist zur 6502 identisch.

BRA - Branch Always

Hierbei handelt es sich um einen Verzweigungsbefehl, der sich prinzipiell genauso verhält, wie BNE, BEQ usw. Nur wird hier keine Bedingung abgetestet.

BRL - Branch Long

Hierbei handelt es sich wie bei BRA auch um einen unconditional Branch. Nur gibt es hier einen 16Bit vorzeichenbehafteten Offset. Es sind demzufolge Verzweigungen von -32768 bis +32767 möglich. Da es sich Bei der Addition des 16Bit Offsets zum Program Counter um eine Modulo 65536 Operation handelt (also nicht über Bankgrenzen hinweg verzweigt wird), kann damit innerhalb einer Bank an jede beliebige Stelle verzweigt werden.

BRK - Force Break

Das ist eigentlich kein neuer Befehl. Aber der neue BRK ist um ein zusätzliches Byte erweitert worden: um ein Signature Byte. Diese Signature kann z.B. dazu genutzt werden, um dem Betriebssystem noch spezielle Informationen mitzuteilen, wie der Break behandelt werden soll. Da das zusätzliche Byte praktisch nur als Dummy hinter dem Break steht und nicht von der CPU ausgewertet wird, gibt es keine Inkompatibilitäten mit vorhandenen Programmen. Das Signature Byte kann über die wie üblich auf dem Stack abgelegte Adresse des Breaks ausgewertet werden. Außerdem gibt es im Native Mode für den Break Interrupt einen eigenen Interruptvektor (siehe Seite 63).

COP - Coprozessor

Dieser Befehl war von WDC für die Einbindung eines Coprozesors gedacht. COP arbeitet exakt nach dem selben Prinzip wie BRK, es wird nur durch einen eigenen COP-Vector gesprungen (siehe Seite 63). WDC hat die COP-Signaturen $80 - $FF für zukünftige Befehle eines Coprozessors reserviert. Die Signaturen $00 - $7F sind zur freien Benutzung freigegeben. Auf meine Anfrage hin, wurde mir von WDC mitgeteilt das von einem evtl. Coprozessor derzeit noch nichts spezifiziert ist. Trotzdem sollten die reservierten Signaturen unbenutzt bleiben.

DEC A, INC A

Damit kann der Accu direkt dekrementiert bzw. inkrementiert werden.

JML - Jump Long

Das ist der einzige direkte Jump, mit dem es möglich ist die Bank des aktuellen Programmcodes zu wechseln.

Tabelle B.1: Alphabetisch geordneter Befehlssatz

ADC	Add Memory to Accumulator with Carry	PHB	Push Data Bank Register on Stack
AND	'AND' Memory with Accumulator	PHD	Push Direct Register on Stack
ASL	Shift One Bit Left, Memory or Accumulator	PHK	Push Program Bank Register on Stack
BCC	Branch on Carry clear (Pc=0)	PHP	Push Processor Status on Stack
BCS	Branch on Carry set (Pc=1)	PHX	Push Index X on Stack
BEQ	Branch if Equal (Pz=1)	PHY	Push index Y on Stack
BIT	Bit Test	PLA	Pull Accumulator from Stack
BMI	Branch if Result Minus (Pn=1)	PLB	Pull Data Bank Register from Stack
BNE	Branch if Not Equal (Pz=0)	PLD	Pull Direct Register form Stack
BPL	Branch if Result Plus (Pn=1)	PLP	Pull Processor Status from Stack
BRA	Branch Always	PLX	Pull Index X from Stack
BRK	Force Break	PLY	Pull Index Y from Stack
BRL	Branch Always Long	REP	Reset Processor Status
BVC	Branch on Overflow Clear (Pv=0)	ROL	Rotate One Bit Left (Memory or Accumulator)
BVS	Branch on Overflow Set (Pv=1)	ROR	Rotate One Bit Right (Memory or Accumulator)
CLC	Clear Carry Flag	RTI	Return From Interrupt
CLD	Clear Decimal Mode Flag	RTL	Return From Subroutine Long
CLI	Clear Interrupt Disable Flag	RTS	Return from Subroutine
CLV	Clear Overflow Flag	SBC	Subtract Memory from Accumulator with Borrow
CMP	Compare Memory with Acumulator	SEP	Set Processor Status
COP	Coprocessor	SEC	Set Set Carry Flag
CPX	Compare Memory and Index X	SED	Set Decimal Mode Flag
CPY	Compare Memory and Index Y	SEI	Set Interrupt Disable Flag
DEC	Decrement Memory or Accumulator by One	STA	Store Accumuator in Memory
DEX	Decrement Index X by One	STP	Stop the Clock
DEY	Decrement Index Y by One	STX	Store Index X in Memory
EOR	'Exclusive OR' Memory with Accumulator	STY	Store Index Y in Memory
INC	Increment Memory or Accumulator by One	STZ	Store Zero in Memory
INX	Increment Index X by One	TAX	Transfer Accumulator to Index X
INY	Increment Index Y by One	TAY	Transfer Accumulator to Index Y
JML	Jump Long (across Bank Boundaries)	TCD	Transfer C Accumulator to Direct Register
JMP	Jump to New Location (Bank relative)	TCS	Transfer C Accumulator to Stack Pointer Register
JSL	Jump to Subroutine (across Bank Boundaries)	TDC	Transfer Direct Register to C Accumulator
JSR	Jump to Subroutine (Bank relative)	TRB	Test and Reset Bit
LDA	Load Accumulator with Memory	TSB	Test and Set Bit
LDX	Load Index X with Memory	TSC	Transfer Stack Pointer to C Accumulator
LDY	Load Index Y with Memory	TSX	Transfer Stack Pointer to Index X
LSR	Shift One Bit Right (Memory or Accumulator)	TXA	Transfer Index X to Accumulator
MVN	Block Move Negative	TXS	Transfer Index X to Stack Pointer
MVP	Block Move Positive	TXY	Transfer Index X to Index Y
NOP	No Operation	TYA	Transfer Index Y to Accumulator
ORA	'OR' Memory with Accumulator	TYX	Transfer Index Y to Index X
PEA	Push Effektive Absolute Address on Stack (or Pus Immediate Data on Stack)	WAI	Wait for Interrupt
PEI	Push Effective Indirect Address on Stack (or Push Direct Data on Stack)	WDM	Reserved for future use
PER	Push Effektive Program Counter Relative on Stack	XBA	Exchange B and A Accumulator
PHA	Push Accumulator on Stack	XCE	Exchange Carry and Emultion Bits

JSL - Jump to Subroutine Long

Das ist der einzige direkte Jump, mit dem es möglich ist, ein Unterprogramm in einer anderen Bank aufzurufen. Dabei wird eine 24Bit Adresse auf dem Stack abgelegt.

MVN und MVP - Block Move Negative/Positive

Diese Befehle sind schon weitgehend auf Seite 58 beschrieben. *MVN* und *MVP* unterscheiden sich in ihrer Anwendung in der Hinsicht, daß *MVN* benutzt wird, wenn die Ziel-Startadresse kleiner als die Quell-Startadresse ist. Bei *MVP* verhält es sich genau umgekehrt. Das ist insbesondere dann zu beachten, wenn sich Ziel- und Quellbereich überlappen.
Bei einem *MVP* müssen die Indexregister zu Beginn mit der oberen Quell- bzw. Zieladresse geladen werden, bei einem *MVN* mit der unteren.

PEA - Push Effective Absolute Address or 16 Bit Immediate Data on Stack

Mit diesem Befehl ist es möglich gleich einen 16Bit Immediate Wert (der unmittelbar auf den OpCode folgt) auf dem Stack abzulegen. Damit entfällt der Weg dies über ein Register zu erledigen. Der Wert kann vom Programmierer entweder als 16Bit Adresse oder als normales Datum betrachtet werden. Damit können zum Beispiel in Verbindung mit der Adressierung *d,s* bzw. *(d,s),y* direkt Daten bzw. Zeiger auf Arrays übergeben werden.

PEI - Push Indirect Address or Direct Data on Stack

Dieser Befehl ist eine Kombination aus der Direct- und Stackadressierung. Das zweite Byte des Befehls und das Direct Register ergeben eine 16Bit Adresse in Bank $00. Aus dieser Addresse wird ein 16Bit Word gelesen und dann auf dem Stack abgelegt. Mit *PEI* ist also das selbe möglich, wie mit *PEA*, nur mit dem Vorteil, daß flexible Pointer bzw. Daten übergeben werden können.

PER - Push Effektive Program Counter Relative on Stack

Dem OpCode folgt ein 16Bit Offset. Die Addition dieses Offsets und des Program Counters wird auf dem Stack abgelegt.

PHB, PHD, PHK, PHX, PHY, PLB, PLD, PLX, PLY

Diese Befehle dürften sich selbst erklären...

REP - Reset Processor Status

Dieser Befehl hat zur Folge, daß bei allen Bits, die im Immediate-Byte gesetzt sind, die entsprechenden Bits im Prozessor Status P gelöscht werden.
Ein *REP #%00110000* hat demnach zur Folge, daß die Bits M und X in P rückgesetzt werden. Alle anderen Bits bleiben unberührt.

RTL - Return from Subroutine Long

RTL holt eine 24Bit Adresse vom Stack und springt dorthin. Damit muß ein mit *JML* aufgerufenes Unterprogramm beendet werden.

SEP - Set Processor Status

SEP arbeitet wie *REP*, nur daß hier die entsprechenden Bits gesetzt werden.

STP - Stop the Clock

Mit *STP* kann die CPU angehalten, oder besser 'abgeschossen' werden. Die einzige Möglichkeit die CPU wieder in einen arbeitsfähigen Zustand zu bringen ist ein Reset.

STZ - Store Zero in Memory

Hiermit kann eine Speicherzelle gelöscht werden. Steht dabei das M-Flag im P auf 0 (also Accu auf 16Bit), werden entsprechend zwei Bytes gelöscht.

TCD, TCS, TDC, TSC, TXY, TYX

Ohne Komentar...

TRB, TSB - Test and Reset/Set Bit

Diese beiden Befehle laufen wie folgt ab:
Je nach dem wie das M-Flag steht, wird ein Word von der angegebenen Adresse geladen. Bei *TRB* wird der Operand anschließend intern mit dem bitweise invertierten Accu 'AND'-verknüpft und in den Speicher zurückgeschrieben. War beim Operanden mindestens ein Bit gesetzt, welches auch im Accu gesetzt ist, wird das Zero Flag gelöscht — ansonsten gesetzt. Bei *TSB* wird mit dem Accu 'OR'-verknüpft und zurückgeschrieben. Das Zero-Flag wird genau wie bei *TRB* behandelt.
Diese beiden Operationen zählen unter anderen zu den Read-Modify-Write (R-M-W) Befehlen. Während eines solchen Befehls wird das Memory Lock Signal (ML) aktiv, welches in Multiprozessor Systemen dazu genutzt werden kann anderen Prozessoren kurzzeitig den Speicherzugriff zu verweigern. Dies ist unbedingt notwendig, um *kritische Abschnitte* korrekt handlen zu können.

WAI - Wait for Interrupt

Mit *WAI* kann die CPU in einen Energiesparmodus versetzt werden. Dieser Befehl zieht die RDY-Leitung auf Low und die CPU arbeitet erst weiter, wenn ein ABORT, NMI, IRQ oder Reset auftritt. Sollte das Interrupt Disable Flag gesetzt sein und es tritt ein IRQ auf, dann wird zwar der Wait Befehl abgebrochen, aber nicht durch den IRQ-Vector gesprungen (die Befehle nach dem *WAI* werden abgearbeitet.

XBA - Exchange B and A Accumulator

Damit werden die beiden Accuhälften vertauscht.

XCE - Exchange Carry and Emulatin Bits

Dieser Befehl dient in erster Linie dazu, zwischen dem Native und Emulation Mode umzuschalten. Die Vorgehensweise ist die, daß erst der Carry gesetzt (zum Schalten in den Emulation Mode) oder gelöscht (Native Mode) wird und dann ein *XCE* ausgeführt wird. Nach dem Umschalten in den Native Mode (und im Emulation Mode) stehen Accu und Index Register auf 8Bit. Aber die neuen Interruptvektoren sind aktiviert!

Tabelle B.2: Interruptvektoren

Emulation Mode (E=1)			Native Mode (E=0)		
$00FFFE,F	IRQ/BRK	Hard/Software	$00FFEE,F	IRQ	Hardware
$00FFFC,D	Reset	Hardware	$00FFEC,D	Reserved	
$00FFFA,B	NMI	Hardware	$00FFEA,B	NMI	Hardware
$00FFF8,9	ABORT	Hardware	$00FFE8,9	ABORT	Hardware
$00FFF6,7	Reserved		$00FFE6,7	BRK	Software
$00FFF4,5	COP	Software	$00FFE4,5	COP	Software

B.4.2 Interrupts

Im Native Mode springt die 65C816 durch andere Interruptvektoren. Das hat den Vorteil, daß somit gleich zu speziellen Routinen verzweigt werden kann. Während der zwei Taktzyklen, wo die Vektoren ausgelesen werden, wird das Vector Pull Signal (VP) aktiv.
Außer dem schon beschriebenem neuen *COP*-Interrupt gibt es noch einen weiteren, den *ABORT*-Interrupt. Tritt ein solcher Interrupt auf, wird im Gegensatz zu allen anderen Interrupts der aktuell laufende Befehl sofort abgebrochen.

B.4.3 OpCodes

Tabelle B.3 zeigt sämtliche OpCodes der 65C816. Die Werte in den OpCode Feldern sind wie folgt zu interpretieren:

Instruction Mnemonic	Addressing Mode
Base Number of Bytes	Base Number of Cycles

B.4.4 Bemerkungen (WICHTIG!!!)

Bytes und Zyklen pro Befehl

- Addiere bei ein Byte (nur bei *Immediate*) und generell einen Zyklus, falls ein 16Bit Operand adressiert wird.
- Addiere bei *Direct*-Referenzierung einen Zyklus, falls niederwertiges Byte von D verschieden von $00.
- Addiere bei *Relative r* einen Zyklus, wenn Verzweigung stattfindet.
- Addiere bei *Relative r* und Emulation Mode einen Zyklus, wenn Verzweigung eine Page Grenze überschreitet.
- Subtrahiere bei Interrupts und *RTI* einen Zyklus, falls im Emulation Mode.

- Addiere bei *(d),y* , *a,x* und *a,y* einen Zyklus, wenn die Bank Grenze überschritten wird. Eigentlich müsste dies auch bei den Adressierungen *[d],y* , *al,x* und *(d,s),y* nötig sein, weil hier der selbe Sachverhalt vorliegt — ist aber in den Datenblättern nicht mit aufgeführt.

Stack Adressierung

Obwohl der Stack sich im Emulation Mode auf den Bereich von $000100 – $0001FF festgelegt ist, werden bei den folgenden Befehlen bzw. Adressierungsarten die Grenzen über- oder unterschritten wenn auf zwei oder drei Bytes zugegriffen wird:
JSL, JSR(a,x), PEA, PEI, PER, PHD, PLD, RTL, d,s , (d,s),y

Direct Adressierung

- Im Emulation Mode befindet sich der Direct Bereich defaultmäßig von $000000 – $0000FF (D steht also auf $0000). D kann aber hier trotzdem mit dem Befehl TCD geändert werden. Dazu muß aber im höherwertigen Byte des Accus vorher auch der gewünschte Wert gebracht werden. Im Emulation Mode ist dies nur mit XBA möglich.

- Ist das niederwertige Byte DL von D im Emulation Mode gleich $00, wird bei Direct Indizierungen (...,x , ...,y) nicht über die Grenze $00DHFF hinwegindiziert. Also zum Beispiel *STA $FF,X* mit X=$01 und D=$0800 ergibt nicht $000900, sondern $000800. Das gilt aber nicht bei den Adressierungsarten *[d]* , *[d],y* und dem Befehl *PEI*.

- Ist das niederwertige Byte von D im Emulation Mode verschieden von $00, wird generell über die oben beschriebene Grenze hinwegindiziert.

Absolute Adressierung

Indizierungen mit den Index Registern X oder Y werden je nach Einstellung mit 8 oder 16Bit ausgeführt. Bei den folgenden Addressierungen findet eine Indizierung der absoluten Adresse durch die Index Register über Bankgrenzen hinweg statt: (d),y , [d],y , a,x , al,x , a,y und (d,s),y

Transfer von 8Bit zu 16Bit Registern und umgekehrt

Bei jedem Transfer von einem in ein anderes Register werden vom Quellregister alle 16Bit bereitgestellt. Wieviele Bits das Zielregister übernimmt, hängt von dessen eingestellter Breite ab. Sind zum Beispiel die Index Register auf 8Bit und der Accu auf 16Bit eingestellt, hat ein *TAX* zur Folge, daß nur das niederwertige Byte des Accus in das niederwertige Byte des X Registers geschrieben werden. Umgekehrt modifiziert also ein *TXA* den kompletten 16Bit Accu. Bei den Transfers *TCS, TSC, TCD und TDC* findet immer ein 16Bit Transfer statt — egal wie der Accu eingestellt ist.

Interrupts

Befindet sich die CPU im Emulation Mode und es tritt irgendein Interrupt auf, dann wird nur der aktuelle Program Counter (also nur eine 16Bit Adresse) auf dem Stack abgelegt. Im Native Mode wird zusätzlich noch das Program Bank Register abgelegt.

Tabelle B.3: OpCode Matrix

MSD	LSD 0	1	2	3	4	5	6	7	MSD
0	BRK s 2 8	ORA (d,x) 2 6	COP s 2 8	ORA d,s 2 4	TSB d 2 5	ORA d 2 3	ASL d 2 5	ORA [d] 2 6	0
1	BPL r 2 2	ORA (d),y 2 5	ORA (d) 2 5	ORA (d,s),y 2 7	TRB d 2 5	ORA d,x 2 4	ASL d,x 2 6	ORA [d],y 2 6	1
2	JSR a 3 6	AND (d,x) 2 6	JSL al 4 8	AND d,s 2 4	BIT d 2 3	AND d 2 3	ROL d 2 5	AND [d] 2 6	2
3	BMI r 2 2	AND (d),y 2 5	AND (d) 2 5	AND (d,s),y 2 7	BIT d,x 2 4	AND d,x 2 4	ROL d,x 2 6	AND [d],y 2 6	3
4	RTI s 1 7	EOR (d,x) 2 6	WDM 2 2	EOR d,s 2 4	MVP xyc 3 7	EOR d 2 3	LSR d 2 5	EOR[d] 2 6	4
5	BVC r 2 2	EOR (d),y 2 5	EOR (d) 2 5	EOR (d,s),y 2 7	MVN xyc 3 7	EOR d,x 2 4	LSR d,x 2 6	EOR [d],y 2 6	5
6	RTS s 1 6	ADC (d,x) 2 6	PER s 3 6	ADC d,s 2 4	STZ d 2 3	ADC d 2 3	ROR d 2 5	ADC [d] 2 6	6
7	BVS r 2 2	ADC (d),y 2 5	ADC (d) 2 5	ADC (d,s),y 2 7	STZ d,x 2 4	ADC d,x 2 4	ROR d,x 2 6	ADC [d],y 2 6	7
8	BRA r 2 2	STA (d,x) 2 6	BRL rl 3 3	STA d,s 2 4	STY d 2 3	STA d 2 3	STX d 2 3	STA [d] 2 6	8
9	BCC r 2 2	STA (d),y 2 6	STA (d) 2 5	STA (d,s),y 2 7	STY d,x 2 4	STA d,x 2 4	STX d,y 2 4	STA [d],y 2 6	9
A	LDY # 2 2	LDA (d,x) 2 6	LDX # 2 2	LDA d,s 2 4	LDY d 2 3	LDA d 2 3	LDX d 2 3	LDA [d] 2 6	A
B	BCS r 2 2	LDA (d),y 2 5	LDA (d) 2 5	LDA (d,s),y 2 7	LDY d,x 2 4	LDA d,x 2 4	LDX d,y 2 4	LDA [d],y 2 6	B
C	CPY # 2 2	CMP (d),y 2 6	REP # 2 3	CMP d,s 2 4	CPY d 2 3	CMP d 2 3	DEC d 2 5	CMP [d] 2 6	C
D	BNE r 2 2	CMP (d),y 2 5	CMP (d) 2 5	CMP (d,s),y 2 7	PEI s 2 6	CMP d,x 2 4	DEC d,x 2 6	CMP [d],y 2 6	D
E	CPX # 2 2	SBC (d,x) 2 6	SEP # 2 3	SBC ds 2 4	CPX d 2 3	SBC d 2 3	INC d 2 5	SBC [d] 2 6	E
F	BEQ r 2 2	SBC (d),y 2 5	SBC (d) 2 5	SBC (d,s),y 2 7	PEA s 3 5	SBC d,x 2 4	INC d,x 2 6	SBC [d],y 2 6	F

MSD	LSD 8	9	A	B	C	D	E	F	MSD
0	PHP s 1 3	ORA # 2 2	ASL A 1 2	PHD s 1 4	TSB a 3 6	ORA a 3 4	ASL a 3 6	ORA al 4 5	0
1	CLC i 1 2	ORA a,y 3 4	INC A 1 2	TCS i 1 2	TRB a 3 6	ORA a,x 3 4	ASL a,x 3 7	ORA al,x 4 5	1
2	PLP s 1 4	AND # 2 2	ROL A 1 2	PLD s 1 5	BIT a 3 4	AND a 3 4	ROL a 3 6	AND al 4 5	2
3	SEC i 1 2	AND a,y 3 4	DEC A 1 2	TSC i 1 2	BIT a,x 3 4	AND a,x 3 4	ROL a,x 3 7	AND al,x 4 5	3
4	PHA s 1 3	EOR # 2 2	LSR A 1 2	PHK s 1 3	JMP a 3 3	EOR a 3 4	LSR a 3 6	EOR al 4 5	4
5	CLI i 1 2	EOR a,y 3 4	PHY s 1 3	TCD i 1 2	JMP al 4 4	EOR a,x 3 4	LSR a,x 3 7	EOR al,x 4 5	5
6	PLA s 1 4	ADC # 2 2	ROR A 1 2	RTL s 1 6	JMP (a) 3 5	ADC a 3 4	ROR a 3 6	ADC al 4 5	6
7	SEI i 1 2	ADC a,y 3 4	PLY s 1 4	TDC i 1 2	JMP (a,x) 3 6	ADC a,x 3 4	ROR a,x 3 7	ADC al,x 4 5	7
8	DEY i 1 2	BIT # 2 2	TXA i 1 2	PHB s 1 3	STY a 3 4	STA a 3 4	STX a 3 4	STA al 4 5	8
9	TYA i 1 2	STA a,y 3 5	TXS i 1 2	TXY i 1 2	STZ a 3 4	STA a,x 3 5	STZ a,x 3 5	STA al,x 4 5	9
A	TAY i 1 2	LDA # 2 2	TAX i 1 2	PLB s 1 4	LDY a 3 4	LDa a 3 4	LDX a 3 4	LDA al 4 5	A
B	CLV i 1 2	LDA a,y 3 4	TSX i 1 2	TYX i 1 2	LDY a,x 3 4	LDA a,x 3 4	LDX a,y 3 4	LDA al,x 4 5	B
C	INY i 1 2	CMP # 2 2	DEX i 1 2	WAI i 1 3	CPY a 3 4	CMP a 3 4	DEC a 3 6	CMP al 4 5	C
D	CLD i 1 2	CMP a,y 3 4	PHX s 1 3	STP i 1 3	JML (a) 3 6	CMP a,x 3 4	DEC a,x 3 7	CMP al,x 4 5	D
E	INX i 1 2	SBC # 2 2	NOP i 1 2	XBA i 1 3	CPX a 3 4	SBC a 3 4	INC a 3 6	SBC al 4 5	E
F	SED i 1 2	SBC a,y 3 4	PLX s 1 4	XCE i 1 2	JSR (a,x) 3 6	SBC a,x 3 4	INC a,x 3 7	SBC al,x 4 5	F

B.4.5 Neues an der Hardware

Die 65C816 besitzt ein paar neue Steuerleitungen nach außen hin, die hier beschrieben werden sollen. Die Anschlußbelegung befindet sich in den HyperSpeed Schaltplänen.

Abort (ABORT)

ABORT wird wie schon erwähnt zum Abbruch von laufenden Operationen verwendet. Ein negativer Übergang verhindert eine Modifikation jeglicher interner Register während des aktuellen Befehls. Dabei wird durch den Vektor $00FFF8,9 (Emulation Mode) bzw. $00FFE8,9 (Native Mode) gesprungen. Ein Abort tritt immer auf, wenn während der Phi2 die *ABORT* Leitung auf Low ist. Daher sollte *ABORT* maximal einen Phi2-Zyklus aktiv sein.

Bus Enable (BE)

Ist *BE* Low, dann werden Adress- und Datenbus der 65C816 hochohmig geschalten. *BE* wird auf HyperSpeed allerdings nur sporalisch verwendet, da hier aufgrund der schwachen Adress- und Datenausgangsbuffer extra Treiber direkt hinter die CPU geschaltet sind.

Data/Address Bus (D0/BA0 – D7/BA7)

Auf diesen 8Bit Bus werden in der ersten Hälfte eines Speicherzyklus die oberen 8Bit der Adresse (A16 – A23) multiplext und in der zweiten Hälfte die Daten. Mit steigender Phi2 muß demnach das höherwertige Byte der Adresse gelatcht werden.

Emulation Status (E)

Auf dieser Leitung wird direkt das Emulation Bit aus dem Processor Status nach außen geführt.

Memory Lock (ML)

Dieses Signal kann dazu genutz werden, um die Integrität von Read-Modify-Write Befehlen (also atomare, nicht zerlegbare Operationen) in Multiprozessorsystemen zu gewährleisten. *ML* ist Low während der letzten drei bzw. fünf Zyklen (ja nach M-Flag) bei den Befehlen *ASL, DEC, INC, LSR, ROL, ROR, TRB* und *TSB* bei Referenzierung des Speichers.

Memory/Index Select Status (M/X)

Über diese Leitung werden die Bits M und X aus dem Prozesser Status multiplext nach außen gebracht. Dabei ist M während der fallenden Phi2 und X während der steigenden Phi2 gültig.

Ready (RDY)

RDY ist hier im Gegensatz zur 6502 ein bidirektionales Signal. Der *WAI* (WAit for Interrupt) Befehl hat zur Folge, daß *RDY* intern auf Low gezogen wird. *WAI* hat aber laut Datenblatt keinen Effekt, wenn *RDY* 'gewaltsam' auf High gezogen wird.
Wird *RDY* extern auf Low gezogen, verweilt die CPU solange im aktuellen Zustand. Ist *RDY* während der Phi2 Low-Phase auf Low, dann wird auf D0/BA0 – D7/BA7 **NICHT** die Bankadresse ausgegeben, sondern im Schreib-Fall die aktuell zu schreibenden Daten. Ein weiterer Unterschied zur 6502 ist, daß die 6502 während Schreibzyklen nicht durch *RDY* angehalten werden kann, die 65C816 aber wohl.

Valid Data Address and Valid Program Adress (VDA and VPA)

Diese beiden Ausgangsleitungen zeigen den aktuellen Zustand einer Befehlsabarbeitung an. Dabei sind die Zustände wie folgt kodiert:

VDA	VPA	
0	0	Interne Operation
0	1	Gültige Programmadresse
1	0	Gültige Datenadresse
1	1	OpCode fetch

VDA und *VPA* stellen sozusagen eine Obermenge an Informationen zur Verfügung, die bei der 6502 die (jetzt entfallene) *SYNC*-Leitung zur Verfügung gestellt hat.

Vector Pull (VP)

Hiermit wird kurz nach einem Interrupt angezeigt (*VP* ist Low), daß die Interruptvektoren von der CPU gelesen werden. Dies geschieht während der letzten zwei Interrupt Zyklen.

B.4.6 Timing

Bild B.1 zeigt das Timing Diagramm der 65C816 und Tabelle B.4 zeigt die entsprechenden Zeitwerte für die 14MHz Version.

Anmerkung: Obwohl für *ABORT* laut Timing Diagramm eine Zeit *tPCS* vor steigender Phi2 eingehalten werden muß, wird ein Abort auch korrekt ausgeführt, wenn *ABORT* erst nach steigender Phi2 aktiv wird. Auf jeden Fall sollte bei *ABORT* aber *tPCS* vor der fallenden Phi2 eingehalten werden.

Abbildung B.1: Timing Diagramm

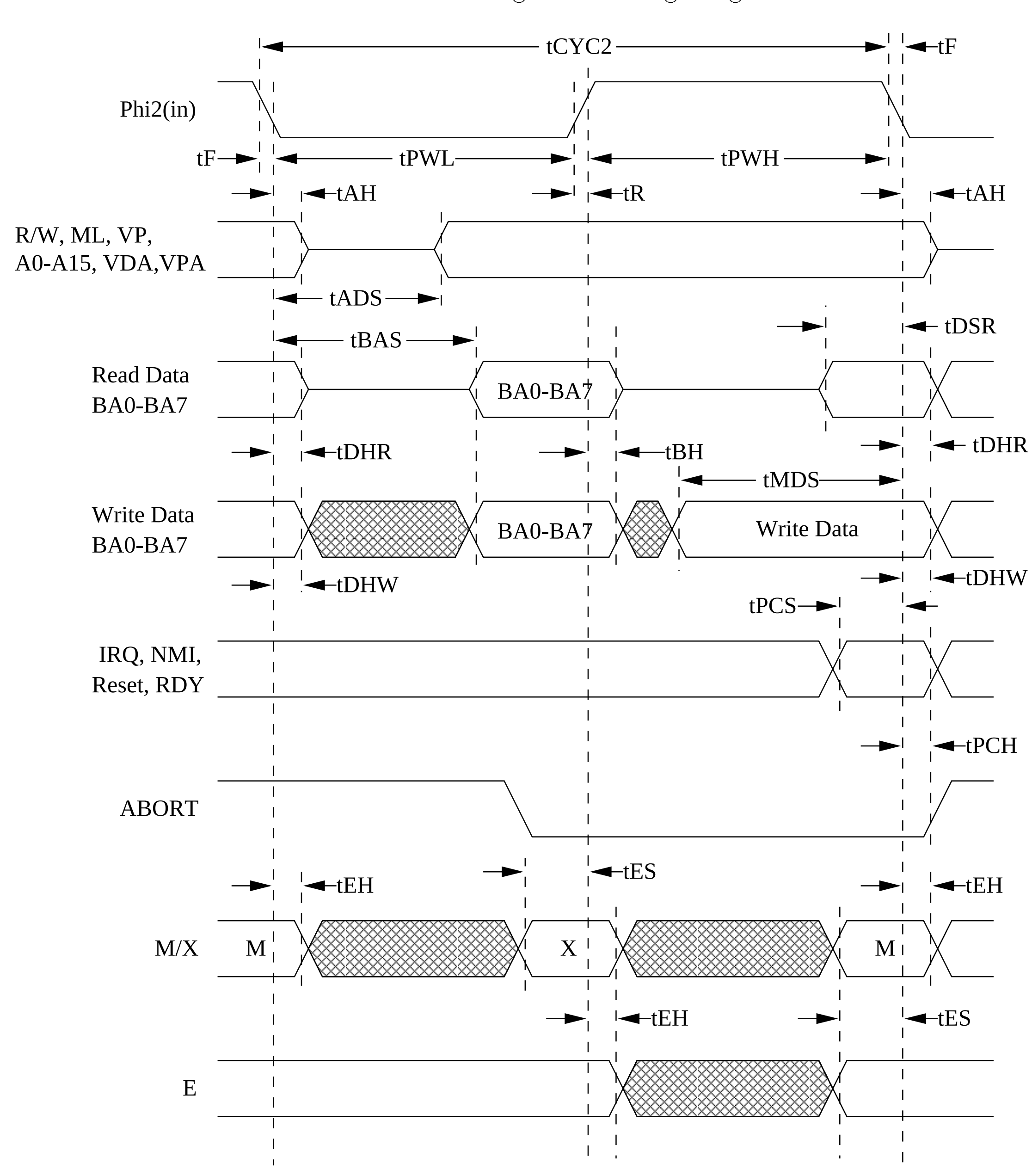


Tabelle B.4: Timing Werte

Parameter	Symbol	14MHz		
		Min	Max	
Cycle Time	tCYC	70	DC	ns
Clock Pulse Width Low	tPWL	35	-	ns
Clock Pulse Width High	tPWH	35	-	ns
Fall Time, Rise Time	tF,tR	-	5	ns
A0–A15 Hold Time	tAH	10	-	ns
A0–A15 Setup Time	tADS	-	30	ns
BA0–BA7 Hold Time	tBH	10	-	ns
BA0–BA7 Setup Time	tBAS	-	33	ns
Read Data Hold Time	tDHR	10	-	ns
Read Data Setup Time	tDSR	10	-	ns
Write Data Delay Time	tMDS	-	30	ns
Write Data Hold Time	tDHW	10	-	ns
Processor Control Setup Time	tPCS	10	-	ns
Processor Control Hold Time	tPCH	10	-	ns
E, MX Output Hold Time	tEH	5	-	ns
E, MX Output Setup Time	tES	10	-	ns

Anhang C

Der Versatile Interface Adapter (VIA) 65C22

Dieser Baustein ist eine Erweiterung des 6520 PIA. Neben den Eigenschaften des PIA besitzt der VIA noch zwei 16Bit Timer und ein bidirektionales serielles Interface (synchron). Das CPU Interface ist das gleiche wie des PIA und soll hier nicht näher beschrieben werden. Für nähere Beschreibungen verweise ich wieder auf die Original Datenblätter.

C.1 Bedeutung der Register des VIA

Tabelle C.1 zeigt alle Register des 65C22.

C.2 Portoperationen

C.2.1 Port A

Alle 8 Bits können wahlweise als Ein- oder Ausgang definiert werden. Dazu dient das Register DDRA. Ist dort ein Bit gesetzt, so ist die entsprechende Portleitung ein Ausgang, andernfalls ein Eingang. Mit Hilfe der *CA1* (Eingangsleitung) können Eingangs-Daten von einem intern Latch gelatcht werden (siehe PCR). Das Signal *CA2* kann wahlweise als Ein- oder Ausgang definiert werden. Über dieses Signal kann auch ein automatisches Handshake bei Read oder Write Operationen erfolgen. Da die Handshake Operationen bei HyperSpeed keine direkte Nutzung finden, sollen diese hier nicht näher behandelt werden. Mittels *CA1* und *CA2* können auch Interrupts ausgelöst werden (siehe PCR, IER).

C.2.2 Port B

Außer einer anderen (hier nicht weiter behandelten) elektrischen Eigenschaft der Leitungen für Port B ist alles wie bei Port A. Lediglich Read-Handshakes können hier nicht ausgelöst werden. Die Leitungen *CB1* und *CB2* können noch zusätzlich von einem seriellen Interface benutzt werden (siehe Shift Register Operation).

Tabelle C.1: 65C22 Register Adressierung

Register	Bez.	Erklärung	
		Write	Read
0	ORB/IRB	Output Register B	Input Register B
1	ORA/IRA	Output Register A	Input Register A
2	DDRB	Data Direction Register B	
3	DDRB	Data Direction Register A	
4	T1C-L	T1 Low-Order Latches	T1 Low-Order Counter
5	T1C-H	T1 High-Order Counter	
6	T1L-L	T1 Low-Order Latches	
7	T1L-H	T1 High-Order Latches	
8	T2C-L	T2 Low-Order Latches	T2 Low-Order Counter
9	T2C-H	T2 High-Order Counter	
A	SR	Shift Register	
B	ACR	Auxiliary Control Register	
C	PCR	Peripheral Control Register	
D	IFR	Interrupt Flag Register	
E	IER	Interrupt Enable Register	
F	ORA/IRA	wie Reg.1, aber ohne Handshake	

C.3 Diverse Steuerregister

C.3.1 Reg.C — Peripheral Control Register PCR

Die niederwertigen 4 Bit kontrollieren die Steuerleitungen für Port A und die höherwertigen 4 Bit entsprechend für Port B.
Bit 0 legt die Latch/Interrupt Eingangsflanke von *CA1* fest. 0 bedeutet dabei, daß die Latch/Interrupt Logik auf eine fallende Flanke reagiert und 1 auf eine steigende. Das gleiche gilt für das Steuerbit von *CB1*.
Die Bits 1, 2 und 3 steuern die Funktionalität von *CA2*. Dabei gibt es folgende Belegungen:

3	2	1	Operation
0	0	0	Input Negative Active Edge
0	0	1	Independent Int.; Input Negative Edge
0	1	0	Input Positive Active Edge
0	1	1	Independent Int.; Input Positive Edge
1	0	0	Handshake Output (Low Active)
1	0	1	Pulse Output (Low Active)
1	1	0	Low Output
1	1	1	High Output

Die Belegungen für *CB2* sind genauso, nur die betreffenden Bits sind dann 7, 6 und 5.
Sind die Leitungen als 'Independent Interrupt' deklariert, bedeutet das, daß sie unabhängig von Portoperationen arbeiten. Mehr dazu siehe IFR.

C.3.2 Reg.B — Auxiliary Control Register ACR

Hier werden einige durcheinandergewürfelte Funktionen gesetzt. Bit 0 legt fest, ob bei Portoperationen auf Port A das interne Latchen von Eingangsdaten mittels *CA1* aktiviert (1) oder nicht aktiviert (0) ist. Bit 1 wirkt entsprechend auf Port B.
Die Bits 4, 3 und 2 steuern das Schieberegister mit folgenden Funktionen:

4	3	2	Shift Reg. Operation
0	0	0	Disabled
0	0	1	Shift in under Control of T2
0	1	0	Shift in under Control of Phi2
0	1	1	Shift in under Control of Ext.Clk (CB1)
1	0	0	Shift out Free Running at T2 Rate
1	0	1	Shift out under Control of T2
1	1	0	Shift out under Control of Phi2
1	1	1	Shift out under Control of Ext.Clk (CB1)

Bit 5 (T2 Timer Control) steuert einen speziellen Modus des zweiten Timers. Ist das Bit=1, wird mit jedem Pulse an PB6 heruntergezählt. Ist das Bit=0, läuft der Timer im normalen 'Timed Interrupt' Modus.
Bit 7 und 6 steuern auch einen speziellen Modus von Timer 1 und sind wie folgt belegt:

7	6	Timer 1 Operation	PB7 Output
0	0	Timed Int. each Time T1 is loaded	Disabled
0	1	Continuous Interrupts	Disabled
1	0	Timed Int. each Time T1 is loaded	One Shot Output
1	1	Continuous Interrupts	Square Wave Output

Nähere Erklärungen befinden sich bei den entsprechenden Einheiten.

C.3.3 Reg.D — Interrupt Flag Register IFR

Hier werden sämtliche Interrupts des VIAs registriert. Ist das entsprechende Bit gesetzt, kam der IRQ von der entsprechenden Einheit. Die einzelnen Bits (bis auf Bit 7) können entweder durch Schreiben in das IFR oder durch entsprechende Aktionen gelöscht werden. Sollte sich *CA2* bzw. *CB2* im Independent Mode befinden, so werden diese Bits nicht durch entsprechende Portoperationen gelöscht. Sie müssen in diesem Fall direkt im IFR gelöscht werden. Bit 7 zeigt an, ob der IRQ allgemein vom VIA kam. Dieses Bit kann nur gelöscht werden, indem alle einzelnen Interruptquellen zurückgesetzt werden.

Bit	Set by	Cleared by
0	CA2 Active Edge	Read or Write ORA
1	CA1 Active Edge	Read or Write ORA
2	Complete 8 Shifts (SR)	Read or Write SR
3	CB2 Active Edge	Read or Write ORB
4	CB1 Active Edge	Read or Write ORB
5	Time Out of T2	Read T2C-L or Write T2C-H
6	Time Out of T1	Read T1C-H /T1L-H or Write T1C-L
7	Any enabled Int.	Clear all Interrupts

C.3.4 Reg.E — Interrupt Enable Register IER

In diesem Register wird für jede Einheit festgehalten, ob sie einen IRQ erzeugen darf (1= Interrupt enabled). Die Bitbelegung von IER entspricht dabei bis auf Bit 7 der von IFR. Beim Schreiben auf dieses Register ist folgendes zu beachten:
Ist Bit 7 im geschriebenen Byte eins, werden die Bits in IER gesetzt, bei denen das entsprechende Bit (Bit 0-6) gesetzt ist. Ist Bit 7 dagegen null, werden die entsprechenden Bits in IER gelöscht. Das Manipulieren von IER ist also mit einem *SEP* oder *REP* bei der 65C816 zu vergleichen. Wird IER ausgelesen, werden alle Bits normal ausgegeben. Bit 7 wird dabei immer als 1 gelesen.

C.4 Timer 1 Operation

Unmittelbar nachdem der Counter geladen wurde (nach einem Write in T1C-H), wird er mit dem VIA-Takt (Phi2 bzw. bei HyperSpeed HPhi2) heruntergezählt. Unterschreitet der Timer den Wert 0 (also falls er auf $FFFF steht), wird das entsprechende Interrupt Flag gesetzt und der IRQ Ausgang des VIAs geht auf Low falls der Interrupt Enabled wurde. Falls das Bit 6 in ACR gesetzt ist, wird automatisch der 16Bit Wert im Latch (T1L-L/H) in den Counter übernommen und wieder von vorn begonnen (Free Run Mode).
Sollte Bit 7 in ACR gesetzt sein, wird bei kontinuierlichen Interrupts von Timer 1 bei jedem Erreichen von 0 der Ausgang auf PB7 invertiert (Square Wave Output Mode). Bei einmaligen Interrupts, also nur wenn der Timer jedesmal von Hand neu geladen werden muß, geht der Ausgang auf PB7 unmittelbar nachdem T1 gestartet wurde auf Low, und geht wieder auf High, wenn T1 0 erreicht hat (One-Shot Output Mode). Achtung: Um eine Ausgabe auf dem PB7 Pin zu ermöglichen, muß PB7 auch auf Ausgabe in DDRB eingestellt sein. Das Interrupt Flag von Timer 1 in IFR wird gelöscht, falls in T1C-H oder T1L-H geschrieben oder T1C-L ausgelesen wird.

C.5 Timer 2 Operation

Der zweite Timer arbeitet nur im One-Shot Mode, d.h. unmittelbar nachdem in T2C-H geschrieben wurde wird T2 gestartet. Unterschreitet T2 0, wird auch das entsprechende Interrupt Flag gesetzt und IRQ geht auf Low. T2 bleibt aber nicht stehen, sondern zählt bei $FFFE, $FFFD, ... weiter. Das Interrupt Flag kann durch Lesen von T2C-L oder Schreiben in T2C-H wieder rückgesetzt werden.
Weiterhin kann als Takt für T2 ein externer Takt verwendet werden (PB6). Dabei wird PB6 intern mit der führenden Flanke der Phi2 gelatcht und als Takt an T2 weitergegeben.

C.6 Shift Register Operation

Mit dem 8Bit Schieberegister wird ein synchrones serielles Interface zur Verfügung gestellt. Dabei wird die *CB2*-Leitung als Datenleitung und die *CB1*-Leitung als Takt verwendet. Bei Schiebeoperationen wird das MSB (Bit 7) in SR zuerst heraus- bzw. hereingeschoben. Im ACR werden die einzelnen Modi eingestellt.

C.6.1 SR Mode 0 — Shift Register Interrupt Disabled

Hier wird zwar bei jeder steigenden Flanke an *CB1* der aktuell an *CB2* anliegende Wert in SR eingschoben, aber es werden keine Interrupts ausgelöst.

C.6.2 SR Mode 1 — Shift in under Control of T2

Hier wird der Schiebetakt durch die niederwertigen 8 Bit des zweiten Timers kontrolliert.
Eine Schiebeoperation wird ausgelöst, durch ein Lesen oder Schreiben von SR falls das SR-Flag im IFR gesetzt ist. Der erste Takt auf *CB1* erfolgt beim nächsten Time Out von T2. Das Bit, welches zu diesem Zeitpunkt auf *CB2* liegt, wird mit dem folgendem Phi2 Zyklus in SR eingeschoben. Nach 8 *CB1*-Takten wird das SR-Flag in IFR gesetzt und ein IRQ ausgelöst.

C.6.3 SR Mode 2 — Shift in under Phi2 Control

Mit einem Read oder Write auf SR wird wieder die Operation gestartet. Im unmittelbar folgendem Zyklus wird *CB1* auf Low gesetzt und im darauf folgendem wieder auf High. Damit werden auch die Daten eingeschoben. Die Schieberate beträgt also die Hälfte des VIA-Taktes. Nach 8 Schiebungen wird wieder ein IRQ ausgelöst.

C.6.4 SR Mode 3 — Shift in under Control of CB1

Hier wird *CB1* als Eingang konfiguriert und mit jeder steigenden Flanke wird wieder *CB2* eingeschoben und nach 8 Schiebungen ein IRQ ausgelöst. Dabei muß *CB2* einen vollen VIA-Taktzyklus nachdem *CB1* auf High gegangen ist stabil bleiben.

C.6.5 SR Mode 4 — Shift out under T2 Control (Free-Run)

Das Schieberegister arbeitet nahezu genauso wie in Mode 1, nur als Ausgang. Aber hier wird nicht nach 8 Schiebungen abgebrochen, sondern permanent der Wert in SR über *CB2* herausgeschoben. Ist Bit 7 von SR herausgeschoben, geht es bei Bit 0 wieder weiter.

C.6.6 SR Mode 5 — Shift out under T2 Control

Wie Mode 5, hier wird aber nach 8 Schiebnungen ein IRQ ausgelöst und das Shifting disabled. Nach einem Read oder Write auf SR wird die Schiebeoperation ausgelöst.

C.6.7 SR Mode 6 — Shift out under Phi2 Control

Wie Mode 2, nur Ausgang.

C.6.8 SR Mode 7 — Shift out under CB1 Control

Dieser Mode arbeitet ähnlich Mode 3. Nach 8 Ausschiebungen wird wieder ein Interrupt ausgelöst, allerdings wird hier das Schieberegister nicht disabled. Bei jedem Read oder Write auf SR wird das SR-Interrupt Flag gelöscht, der SR-Counter wird initialisiert und es werden die nächsten 8 Takte auf *CB1* gezählt und entsprechend ausgeschoben. Anschließend wird wieder ein IRQ ausgelöst usw....

Anhang D

ispLSI Listings

```
// Thu May 02 19:55:19 1996
// D:\PDS2\MMU4_3.LDF generated using Lattice pDS Version 2.8

LDF 1.00.00 DESIGNLDF;
DESIGN MMU;
REVISION 4.0;
AUTHOR Digital Force / Mario Trams;
PROJECTNAME HyperSpeed XL/XE V2.1;
DESCRIPTION
MMU ist selbst Memory Mapped und bentigt keine extra-
Steuerleitungen mehr.
Entgueltige Version!;

PART ispLSI2032-110LJ44;

OPTION PULLUP OFF;

OPTION ISP_EXCEPT_Y2 ON;

OPTION ISP ON;

DECLARE
END; //DECLARE

SYM GLB  A7  1  ;
sigtype RAM1 critical out;
sigtype RAM2 critical out;
sigtype RAM3 critical out;
sigtype ROM critical out;

equations

Bank0=!BA7 & !BA6 & !BA5 & !BA4 & !BA3 & !BA2 & !BA1 & !BA0;

//RAM1 Enable (high-active); Bank $00-$01 fuer 128kx8 RAM
//In Bank $00 ist RAM1 nur aktiv, wenn weder auf den ATARI, noch auf den neuen
//ROM zugegriffen wird (ROM wird hier direkt substituiert)
RAM1 = !BA7 & !BA6 & !BA5 & !BA4 & !BA3 & !BA2 & !BA1 & BA0 & ExtSel
  # Bank0 & !Bank16K & !IO_Range & !ROM & ExtSel;

//RAM2 enable (high-active); Bank $02-$03
RAM2=!BA7 & !BA6 & !BA5 & !BA4 & !BA3 & !BA2 & BA1 & ExtSel;

//RAM3 enable (high-active); Bank $04-$05
RAM3 =!BA7 & !BA6 & !BA5 & !BA4 & !BA3 & BA2 & !BA1 & ExtSel;

//ROM Enable (high-active)
ROM =  BA7 & BA6 & BA5 & BA4 & ExtSel //Zugriff auf hohen ROM-Bereich (Bank $F0-$FF)
         # Bank0 & Bank0_ROM & !IO_Range & ExtSel; //evtl. ROM-Zugriff auf $00C000-$00FFFF

end;
END;

SYM GLB  A6  1  ;
sigtype New_IO critical out;
sigtype ATARI  critical out;
sigtype Bank0 critical out;

equations

Bank0=!BA7 & !BA6 & !BA5 & !BA4 & !BA3 & !BA2 & !BA1 & !BA0;
BankEF=BA7 & BA6 & BA5 & !BA4 & BA3 & BA2 & BA1 & BA0;
//NEW_IO=1 (intern High aktiv) => Zugriff auf neuen IO_Bereich ($EF0000-$EF1FFF)
New_IO = BA7 & BA6 & BA5 & !BA4 & BA3 & BA2 & BA1 & BA0 & !A15 & !A14 & !A13;

//ATARI=1 (hier High-aktiv) => Zugriff auf ATARI-Speicherbereich
//Zugriff wird sofort wieder entzogen, falls ein Abort von der
//Hardware Protection Unit vorliegt
//IO_Range wird von $EFD000-$EFD7FF gespiegelt (an HPU vorbei)
ATARI= (Bank0 & Bank16K # (Bank0 # BankEF) & IO_Range) & !ABORT;

end;
END;

SYM GLB  A4  1  ;
sigtype ABORT critical out;
sigtype Bit7_0 critical reg out;
sigtype Bit3_0 critical reg out;
sigtype Bit2_0 critical reg out;


equations

Bit7_0.clk = Phi1;
Bit3_0.clk = Phi1;
Bit2_0.clk = Phi1;

Bit3_0.d = !CS & !A0 & !RW & BAD3 # (CS # A0 # RW) & Bit3_0.q;
Bit2_0.d = !CS & !A0 & !RW & BAD2 # (CS # A0 # RW) & Bit2_0.q;
//Bit7_0 entspricht ROM_OS; nach HPU-ABORT soll ROM aktiv sein => bei Abort loeschen
Bit7_0.d = (!CS & !A0 & !RW & BAD0 # (CS # A0 # RW) & Bit7_0.q) & !ABORT.pin;

//Bitzuordnungen
HW_W_Protect = Bit0_1;
HW_R_Protect = Bit1_1;

//ABORT wird aktiv (intern High), wenn ein entsprechender Datenzugriff auf den
//Hardwareregisterbereich des ATARIs zugegriffen wird.
ABORT = VDA & Bank0 & IO_Range & (HW_W_Protect & !RW # HW_R_Protect & RW);

end;
END;

SYM GLB  A3  1  ;
sigtype Bank16K out;
sigtype IO_Range critical out;
sigtype Bank0_ROM critical out;

equations;

//Hier findet eine Vordekodierung der unteren 16Bit Adresse statt.
//Alles was hier ausdekodiert wird, laeuft unter der Annahme,
//dass auf Bank $00, also auf $00XXXX zugegriffen wird.

//Steuerbit-Zuordnungen
ATARI_Mem_0     = Bit0_0;
ATARI_Mem_1     = Bit1_0;
ATARI_Mem_2_0 = Bit2_0;
ATARI_Mem_2_1 = Bit3_0;
ATARI_Mem_3     = Bit4_0;
ATR_Mode           = Bit5_0;
ATARI_IO              = Bit6_0;
ROM_OS               = Bit7_0;

//ATARI_Mem_0=1 => von $000000-$003FFF ATARI-Speicher
Bank16K_0= !A15 & !A14 & ATARI_Mem_0 & ExtSel;

//ATARI_Mem_1=1 oder ATARI RAMDisk aktiv und ATARI Mode aktiviert
//=> von $004000-$007FFF ATARI-Speicher
Bank16K_1= !A15 &  A14 & ExtSel & (ATARI_Mem_1 # !ATR_RD & ATR_Mode);

//Bereich von $8000-$BFFF noch mal unterteilt
//ATARI_Mem_2_0=1 => von $008000-$009FFF ATARI-Speicher
//ATARI_Mem_2_1=1 => von $00A000-$00BFFF ATARI-Speicher
Bank16K_2=  A15 & !A14  & !A13 & ATARI_Mem_2_0 & ExtSel
         # A15 & !A14 &   A13 & ATARI_MEM_2_1 & ExtSel;

//ATARI_Mem_3=1 => von $00C000-$00FFFF ATARI-Speicher
Bank16K_3=  A15 &  A14 & ATARI_Mem_3 & ExtSel;

//Zusammenfassung der einzelnen Stuecke:
//Bank16K=1 =>  moeglicher ATARI-Zugriff
Bank16K=Bank16K_0 # Bank16K_1 # Bank16K_2 # Bank16K_3;

//Anzeige, ob Zugriff auf den ATARI Hardwarebereich von $00D000-$00D7FF
//Zugriff wird stattgegeben, falls ATARI_IO=1 oder falls ohnehin der gesamte
//Bereich von $00C000-$00FFFF eingeblendet ist
IO_Range= A15 &  A14 & !A13 & A12 & !A11 & ATARI_IO & ExtSel
# A15 &  A14 & !A13 & A12 & !A11 & ATARI_Mem_3 & ExtSel;

//Vordekodierung, ob moeglicherweise Zugriff auf neuen ROM in Bank0
//ATR_Mode=0 und ATARI_Mem_3=0 => Steuerung durch ROM_OS:
//   ROM_OS=0 => von $00C000-$00FFFF neuer ROM
//ATR_Mode=1 und ATARI_Mem_3=1 => Steuerung durch OS-Bit des ATARIs:
//   ATR_OS=0 => neuer RAM
//   ATR_OS=1 => neuer ROM
//ATARI_Mode_3=1 => ohnehin kein neuer Speicher
Bank0_ROM = A15 & A14 & !ATARI_Mem_3 & !ROM_OS & !ATR_Mode
     # A15 & A14 & !ATARI_Mem_3 & !ATR_Mode & ATR_OS;


end;
END;

SYM GLB  A0  1  ;
sigtype Bit1_0 critical reg out;
sigtype Bit0_0 critical reg out;
sigtype Bit0_1 critical reg out;
sigtype Bit1_1 critical reg out;

equations

//Output Enable aktiv, falls lesend auf das erste Register zugegriffen wird
output.oe = Phi2_CPU & !CS & A0 & RW;

Bit1_0.clk = Phi1;
Bit0_0.clk = Phi1;
```

```
Bit0_1.clk = Phi1;
Bit1_1.clk = Phi1;


Bit1_0.d = !CS & !A0 & !RW & BAD1 # (CS # A0 # RW) & Bit1_0.q;
Bit0_0.d = !CS & !A0 & !RW & BAD7 # (CS # A0 # RW) & Bit0_0.q;

Bit0_1.d = (!CS & A0 & !RW & BAD0 # (CS # !A0 # RW) & Bit0_1.q);
Bit1_1.d = (!CS & A0 & !RW & BAD1 # (CS # !A0 # RW) & Bit1_1.q);

end;
END;

SYM GLB  A1  1  ;
sigtype BA1 critical out;
sigtype BA0 critical out;
sigtype BA7 critical out;
sigtype BA6 critical out;

equations

BA1 = BA1.pin & (Phi2_CPU # !RDY) # BAD1 & !Phi2_CPU & RDY;
BA0 = BA0.pin & (Phi2_CPU # !RDY) # BAD0 & !Phi2_CPU & RDY;
BA7 = BA7.pin & (Phi2_CPU # !RDY) # BAD7 & !Phi2_CPU & RDY;
BA6 = BA6.pin & (Phi2_CPU # !RDY) # BAD6 & !Phi2_CPU & RDY;

end;
END;

SYM GLB  A5  1  ;
sigtype Bit5_0 critical reg out;
sigtype Bit4_0 critical reg out;
sigtype Bit2_1 reg out;
sigtype Bit6_0 critical reg out;

equations

Bit5_0.clk = Phi1;
Bit4_0.clk = Phi1;
Bit6_0.clk = Phi1;
Bit2_1.clk = Phi1;

//Bit4_0 und Bit5_0 entsprechen ATARI_Mem_3 bzw. ATR_Mode und sollen nach
//MPU-ABORT disabled sein (garantiert neuer ROM nach MPU-ABORT)
Bit5_0.d = (!CS & !A0 & !RW & BAD5 # (CS # A0 # RW) & Bit5_0.q) & !ABORT;
Bit4_0.d = (!CS & !A0 & !RW & BAD4 # (CS # A0 # RW) & Bit4_0.q) & !ABORT;

Bit6_0.d = !CS & !A0 & !RW & BAD6 # (CS # A0 # RW) & Bit6_0.q;

//MPU_ABORT bzw. Bit2_1 wird gesetzt, falls durch die
//Hardware Protection Unit ein Abort ausgeloest wurde (ABORT=1)
//und geloescht, falls lesend auf das erste Register zugegriffen wird.
MPU_ABORT = (!Bit2_1.q & ABORT # Bit2_1.q) & !(!CS & A0 & RW);
Bit2_1.d = MPU_ABORT;

end;
END;

SYM GLB  A2  1  ;
sigtype BA3 critical out;
sigtype BA2 critical out;
sigtype BA5 critical out;
sigtype BA4 critical out;

equations

BA3 = BA3.pin & (Phi2_CPU # !RDY) # BAD3 & !Phi2_CPU & RDY;
BA2 = BA2.pin & (Phi2_CPU # !RDY) # BAD2 & !Phi2_CPU & RDY;
BA5 = BA5.pin & (Phi2_CPU # !RDY) # BAD5 & !Phi2_CPU & RDY;
BA4 = BA4.pin & (Phi2_CPU # !RDY) # BAD4 & !Phi2_CPU & RDY;

end;
END;

SYM IOC  IO0  1  BA/D7;
XPIN IO XBAD7 LOCK 15;
IB11 (BAD7,XBAD7);
END;

SYM IOC  IO1  1  BA/D6;
XPIN IO XBAD6 LOCK 16;
IB11 (BAD6,XBAD6);
END;

SYM IOC  IO2  1  BA/D5;
XPIN IO XBAD5 LOCK 17;
IB11 (BAD5,XBAD5);
END;

SYM IOC  IO3  1  BA/D4;
XPIN IO XBAD4 LOCK 18;
IB11 (BAD4,XBAD4);
END;

SYM IOC  IO4  1  BA/D3;
XPIN IO XBAD3 LOCK 19;
IB11 (BAD3,XBAD3);
END;

SYM IOC  IO5  1  BA/D2;
XPIN IO XBAD2 LOCK 43;
BI11 (BAD2,XBAD2,BIT2_1,output);
END;

SYM IOC  IO6  1  BA/D1;
XPIN IO XBAD1 LOCK 21;
IB11 (BAD1,XBAD1);
END;

SYM IOC  IO16  1  A14;
XPIN IO XA14 LOCK 37;
IB11 (A14,XA14);
END;

SYM IOC  IO17  1  A15;
XPIN IO XA15 LOCK 38;
IB11 (A15,XA15);
END;

SYM IOC  IO7  1  BA/D0;
XPIN IO XBAD0 LOCK 22;
IB11 (BAD0,XBAD0);
END;

SYM IOC  IO8  1  /ExtSel;
XPIN IO XEXTSEL LOCK 26 PULLUP;
IB11 (EXTSEL,XEXTSEL);
END;

SYM IOC  IO10  1  ATR_RD;
XPIN IO XATR_RD LOCK 44;
IB11 (ATR_RD, XATR_RD);
END;

SYM IOC  IO11  1  A0;
XPIN IO XA0 LOCK 25;
IB11 (A0,XA0);
END;

SYM IOC  IO13  1  A11;
XPIN IO XA11 LOCK 30;
IB11 (A11,XA11);
END;

SYM IOC  IO14  1  A12;
XPIN IO XA12 LOCK 31;
IB11 (A12,XA12);
END;

SYM IOC  IO15  1  A13;
XPIN IO XA13 LOCK 32;
IB11 (A13,XA13);
END;

SYM IOC  IO19  1  /ATARI;
XPIN IO XATARI LOCK 40 CRITICAL SLOWSLEW;
OB21 (XATARI,ATARI); //inverted Output
END;

SYM IOC  IO30  1  /RAM2;
XPIN IO XRAM2 LOCK 7 CRITICAL SLOWSLEW;
OB21 (XRAM2,RAM2); //inverted Output
END;

SYM IOC  IO29  1  /ROM;
XPIN IO XROM LOCK 8 CRITICAL SLOWSLEW;
OB21 (XROM,ROM); //inverted Output
END;

SYM IOC  IO28  1  /RAM1;
XPIN IO XRAM1 LOCK 10 CRITICAL SLOWSLEW;
OB21 (XRAM1,RAM1); //inverted Output
END;

SYM IOC  IO25  1  /New_IO;
XPIN IO XNEW_IO LOCK 39 CRITICAL SLOWSLEW;
OB21 (XNEW_IO,NEW_IO); //inverted Output
END;

SYM IOC  IO9  1  ATR_OS;
```

```
XPIN IO XATR_OS LOCK 29;
IB11 (ATR_OS, XATR_OS);
END;

SYM IOC  YO  1  ;
XPIN CLK XPHI1 LOCK 11;
IB11(PHI1, XPHI1);
//XPIN CLK XW LOCK 11;
//IB11 (W, XW);
END;

SYM IOC  IO20  1  /CS;
XPIN IO XCS LOCK 6;
IB11(CS, XCS);
END;

SYM IOC  IO21  1  R/W;
XPIN IO XRW LOCK 42;
IB11(RW, XRW);
END;

SYM IOC  IO22  1  RDY;
XPIN IO XRDY LOCK 28;
IB11(RDY, XRDY);
END;

SYM IOC  IO23  1  Phi2;
XPIN IO XPHI2_CPU LOCK 3;
IB11(PHI2_CPU, XPHI2_CPU);
END;

SYM IOC  IO18  1  /ABORT;
XPIN IO XABORT LOCK 27;
OB21(XABORT, ABORT); //inverted Output
END;

SYM IOC  IO31  1  /RAH3;
XPIN IO XRAH3 LOCK 9 CRITICAL SLOWSLEW;
OB21(XRAH3, RAH3); //inverted Output
END;

SYM IOC  IO24  1  VDA;
XPIN IO XVDA LOCK 4;
IB11(VDA, XVDA);
END;
END;  //LDF DESIGNLDF
```

Anhang E

HyperSpeed PCB

Die Abmessungen der Platine betragen ca. 14x15cm. Am hinteren Ende befinden sich zwei Slots zur Aufnahme von Erweiterungskarten. Standardmäßig ist aber nur der vordere Slot bestückt. Der hintere Slot ist dreireihig ausgeführt wobei dort wiederum die hinteren beiden Reihen 1:1 verbunden sind. Damit ist es möglich an dieser Stelle entweder auch einen Steckkartenaufnehmer oder eine zweireihige Messerleiste zu bestücken. Die Messerleiste muß dann logischerweise in die vorderen beiden Reihen eingelötet werden. Über die Messerleiste kann dann über ein Flachbandkabel eine Erweiterung angeschlossen werden. Allerdings ist diese Möglichkeit keinesfalls zu empfehlen und wenn dann sollte das Flachbandkabel nur 2-3cm lang sein.

Die bei sämtlichen Bustreibern auf HyperSpeed verwendete ABT-Logik (Advanced BiCMOS-Technologie) wird ist eine der jüngsten und besten BiCMOS-Techniken. Diese Logikfamilie besteht derzeit nur aus diversen BusLogik-Bausteinen und ist vom Stromverbrauch mit äquvalenten Bausteinen etwa bei HCT-Logik anzusiedeln (¡5mA) und von der Geschwindigkeit etwa doppelt so schnell wie Fast (ca. 3ns). Zum Vergleich hat die Fast-Logik einen Stromverbrauch von etwa 100mA. Weiterhin können die Ausgänge der ABT-Logik 64mA (Low) bzw. -32mA (High) treiben. Es sei noch einmal gesagt, daß das nur die BusLogik-Schaltkreise betrifft. Ein einzelnes AND-Gatter in Fast-Logik ist auch 3ns schnell.
Bei der DataBridge wird zumindest zum ATARI hin eigentlich nicht derartige High-End Technik benötigt. Aber zur HyperSpeed-Seite hin sollte es doch schon etwas zügiger gehen. Außerdem werden durch die guten elektrischen Eigenschaften Turbulenzen im Power Distribution Network von HyperSpeed (Zuführung von +5V und Ground zu den einzelnen Komponenten) weitestgehend gering gehalten. Trotzdem treten noch genug Spitzen auf. Stellt man sich nur vor, daß alle plötzlich auf Low geschalten werden und maximal belastet werden. Dann fließen dort 8x64mA.

Abbildung E.1: Location der wichtigsten Komponenten